Universitat Politècnica de Catalunya
BarcelonaTech

Master Thesis

Numerical Study of a Heat Exchanger

Author:
Christian ROSSI

Adviser:
Riccardo ROSSI

September 22, 2017

Numerical Study of a Heat Exchanger

Christian Rossi

September 22, 2017

Contents

List of Figures

Abstract

Thermo-fluid dynamic is the science that deals with mass movement and energy transfer. More specifically it involves fluid mechanics and heat transfer and, due to different fields interact between them, the thermo-fluid dynamic problem is defined as *coupled* problem. In this this case there is a dependence between velocity and temperature fields.

From the engineering point of view, the thermo-fluid dynamic problem is very important and very present in standard design and practical analysis. On top of that, the continuing increasing computing power has allowed to solve realistic complex problems in a reliable way and make it really interesting.

In the current work, an application in order to computationally study the thermo-fluid dynamic problem has been developed using the Kratos Multiphysics Software . Objective of this work has been the creation and simulation of an heat exchanger model working in a turbulent regime using the *variational multiscale method* formulation.

First, in order to computationally study the problem, a solver able to study the heat exchange through convection and diffusion has been developed; successively the previous solver has been coupled with the fluid dynamic solver, obtaining a tool in order solve thermally coupled flows, where both physical problems take place in the same domain.

Finally different cases have been analyzed in order to test the reliability of the application and also the information transfer between different subdomains following the Dirichlet-Neumann scheme.

An extra transversal objective has been the competence to getting familiar with Kratos and also to progress with my coding ability using Python and C++.

Resumen

La termo-fluidodinamica es la ciencia que estudia el movimiento de masa y el intercambio de energía. Más específicamente esta rama de la física incorpora la mecánica de fluidos e la la transferencia de calor. Debido a la física del problema, el termo-fluido dinámico se define como problema acoplado porqué diferentes campos iteracionan entre ellos. En el caso in cuestión hay una dependencia entre velocidad y temperatura.

Desde el punto de vista ingenieril, el problema termo-fluido dinamico es muy importante y muy presente en el diseño y en el analisis practico. Además, el continuo avance de potencia de calculo de los ordenadores permite solucionar complejos problemas reales de forma fiable y lo rende interesante.

En este trabajo se ha tratado de desarrollar una aplicación de manera que el problema termo-fluido dinámico fuera accesible bajo el enfoque computacional utilizando el software Kratos Multiphysics. Objetivo has sido simular un modelo de intercambiador de calor en régimen turbulento utilizando la formulación *variational multiscale method* y comparar datos experimentales.

En primer lugar, para poder hacer factible una herramienta de estudio, se ha desarrollado un solver para poder estudiar el intercambio de calor a través de conducción e y difusión; sucesivamente el mismo se ha acoplado con el solver de la aplicación fluido dinámica, obtenido una herramienta en grado de resolver flujos acoplados termicamente, donde los dos campos interaccionan entre ellos en el mismo dominio.

Finalmente se han examinado diferentes casos a fin de evaluar la fiabilidad de la aplicación creada y también la transferencia de informaciones entre sub-dominios siguiendo el esquema Dirichlet-Neumann.

Como objetivo transversal también se ha incluido a la familiarización con el software Kratos y la mejora de las habilidades de programación en Python y C++.

Acknowledgments

First of all I would like to acknowledge my parents, Luciano and Doria, for all the support they provided me during all this long journey; I would not reach to this point if it was not for them.

Secondly, my acknowledge also goes to my thesis' advisor Prof. Riccardo Rossi: the continuous orientation, supervision and monitoring, and also due to the possibility I had in order to join and collaborate with the Kratos team at the CIMNE.

Moreover I would like to thank Ruben Zorrilla and Vicente Mataix for the dedicated time, the material, suggestions and the help i received thank their abilities.

I am also grateful to Sally for supporting me and give me the motivation and the positivity throughout writing this thesis.

Chapter 1

Introduction

1.1 Starting Point of the Thesis

The current project is related with my bachelor degree thesis called *Modelización termofludiodinamica de un intercambiador de calor en regimen tubulento*, carried out at the EUETIB (*Escuela Univeristaria de Ingeniería Tecnica de Barcelona*) and supervised by the professor Joan Grau. During the bachelor thesis I learned how to create a physical model setting all the appropriate conditions, in my case was a thermo-fluid dynamic problem in a turbulent regime, using the finite element software *Elmer* from a user point of view.

The motivation of the current project, in order to apply all the concepts I have been learning during the Numerical Methods master, has been to reproduce the same problem but from a developer point of view using the finite element software *Kratos*.

The interest to improve my computational code skills and also taking advantages of Kratos thermo-fluid dynamic application was not yet implemented, I enthusiastically took the opportunity to join the Kratos team and make that my starting point of the thesis.

On top of that, thanks to the M.Riera Santacreu, whose work provided me the experimental data of my bachelor problem thesis [15], my motivation has been enforced due the chance to compare laboratory tests with my Kratos's application in order to verify its feasibility.

1.2 Thermo-Fluid Dynamic Problem

The thermo-fluid dynamic problem describes the process where heat transfer and fluid flow are involved at the same time. This process is extremely important in engineering and it can be observed in a great variety of practical situations.

Basically all the power production methods involve the heat transfer and fluid

flow as essential problem. The thermo-fluid dynamic problem also governs the heating and air conditioning in buildings and it conditions the pollution of the natural environment. It can also be considered the limiting factor of the electrical machinery design and electronic circuits. Thermo-fluid dynamic process is one of the most important process of the metallurgic industries, where devices such as heat exchangers and condensers are used. So heat transfer and fluid dynamics form the core of many scientific studies and engineering problems.

Thermo-fluid dynamic problem involves two different physical problems and it makes it complex. These kind of problems are called coupled , where both problem's solutions depend on each other.

The thermo-fluid dynamic problem can be studied by two main methods: experimental investigation and numerical calculation. The most reliable results under certain conditions are given by the experimental approach. Most of the time this approach is not feasible, especially for small-scale tests, or prohibitively expensive and often impossible. It is also important to underline the difficulties of the experimental measurement and the inconvenience of the measurement errors.

The numerical approach, thanks also to the availability of super computers, consists of a problem discretization described as a set of differential equations into an algebraic problem. This simplification is what makes the numerical methods powerful and widely used [19]

In the current work the main core has been the implementation of a Kratos application in order to obtain a numerical solution of a heat exchanger model which will be compared with experimental results.

1.3 Objectives

The main objective of this work has been to develop a coupled thermo-fluid dynamic application using Kratos Multipyhisics software. The thermo-fluid dynamic application is composed by two different applications: the computational fluid dynamic application, which was already implemented in Kratos, and the *heat equation application*, whose its implementation has been the first step of this work.

The heat equation application's reliability, once implemented, has been proved by comparing a pure diffusion case, a diffusive dominant case and a convective dominant case with analytical solutions.

In order to control flow instabilities for the convective dominant case, a sub grid scale stabilization method has been implemented too.

The thermo-fluid dynamic application was implemented once the heat equation application was created and it has been tested by comparing the numerical results of the computational model working in a turbulent regime with experimental results.

Extra transversal objectives of this current work have given me the ability to become familiar with Kratos Multiphysics from a developers point of view, which

implied a skill's coding improvement of Python and C++ languages.

Chapter 2

State Of Art

This chapter provides the most important fluid mechanics and heat transfer equations. Also the stabilization techniques and the *variational multiscale method* are introduced and finally an overview of basic concepts of thermodynamics and fluid mechanics are proposed.

2.1 Fluid Mechanics and Heat Transfer Equations

Conservation laws state that a particular measurable property of an isolated physical system does not change as the system evolves over time. Exact conservation laws include *continuity equation, momentum equation and energy equation.*

2.1.1 Continuity Equation

Continuity equation is a conservative law and it is based on the conservation principles where the total mass of a continuum does not change, then:

$$\frac{\partial \rho}{\partial t} + \nabla \cdot (\rho \mathbf{v}) = 0 \tag{2.1}$$

Where ρ is the density and $\mathbf{v}$ is the velocity. The divergence of the mass-density flux $\rho\mathbf{v}$ is also called *transport* term because it transfers quantity from one region to another without making a net contribution over the entire field [5].
For incompressible flows the continuity equations implies:

$$\nabla \cdot \mathbf{v} = 0 \tag{2.2}$$

2.1.2 Momentum Equation

Momentum Equation is a conservative law based on Newton's second law and it is based on the conservation principles where no momentum can be created or destroyed, but merely moves from one place to another. The momentum equation states that *the rate of change of the linear momentum of an arbitrary part of a continuous medium is equal to the resultant force acting on the part in question*, then:

$$\frac{\partial}{\partial t}(\rho \mathbf{v}) + \nabla \cdot (\rho \mathbf{v} \otimes \mathbf{v}) = \nabla \cdot \sigma + \rho \mathbf{g} \tag{2.3}$$

where σ is the stress tensor , $\mathbf{g}$ is the body accelerations (per unit of mass) acting on the continuum and $\otimes$ is the outer product.

And for an incompressible fluid, the momentum equation is:

$$\frac{\partial \mathbf{v}}{\partial t} + (\mathbf{v} \cdot \nabla)\mathbf{v} - \nu \nabla^2 \mathbf{v} = -\frac{1}{\rho}\nabla p + \mathbf{g} \tag{2.4}$$

where ν is the cinematic viscosity and p is the pressure. The term $(\mathbf{v} \cdot \nabla)\mathbf{v}$ is known as the convective term, which makes the Navier-Stokes equation nonlinear, and $\nu \nabla^2 \mathbf{v}$ is know as the viscous term.

2.1.3 Energy Equation

Energy Equation, as the continuity and the momentum equation, is a conservative law and it is based on the conservation principles where no energy can be created or destroyed, but merely moves from one place to another. For an incompressible fluid continuity and momentum equation can determine the flow motion meanwhile, if the fluid is compressible, the energy equation is required. The energy equation is:

$$\rho \left[\frac{\partial h}{\partial t} + \nabla \cdot (h \mathbf{v}) \right] = -\frac{Dp}{Dt} + \nabla \cdot (k \nabla T) + (\tau \cdot \nabla)\mathbf{v} \tag{2.5}$$

where h is the enthalpy, T is the absolute temperature, k is the heat conductivity and $(\tau \cdot \nabla)\mathbf{v}$ is the dissipation function representing the work done against viscous forces and the conduction heat transfer is governed by Fourier's law with being the thermal conductivity.

For compressible flows the *equation of state* is used in order to give the relation between density , pressure and temperature:

$$p = \rho RT \tag{2.6}$$

where R is the gas constant (R = 287.1 J/KgK for air).

For an incompressible fluid the density is considered constant and using the relation $dh = c_p dT$, equation 2.5 takes the next form:

$$\rho c_p \left[\frac{\partial T}{\partial t} + (\mathbf{v} \cdot \nabla) T \right] = \nabla \cdot (k \nabla T) + (\tau \cdot \nabla)\mathbf{v} \tag{2.7}$$

where c_p is the heat capacity at constant pressure.

For incompressible flows the equation of states doesn't exist and the energy equation is not coupled with the continuity and the momentum equations. Due to this condition first it is possible to solve the continuity and the momentum equations in order to obtain the velocity and the pressure distribution without knowing the temperature (the assumption where the fluid properties are not function of the temperature is taken into account). If the density changes due to the temperature variations it is considered in the body force term of the momentum equation. Once the velocity and pressure are obtained, energy equation is solved in order to find the temperature distribution.

2.1.4 Constitutive Equations

Considering a thermo-fluid dynamic problem where it is described as

$$\begin{cases} \frac{D\rho}{Dt} + \rho \nabla \cdot \mathbf{v} = 0 \\ \rho \frac{\partial \mathbf{v}}{\partial t} + \rho(\mathbf{v} \cdot \nabla)\mathbf{v} = \nabla \cdot \sigma + \rho \mathbf{g} \\ \rho \frac{\partial e}{\partial t} + \rho \mathbf{v} \cdot \nabla e = \sigma : \nabla \mathbf{v} - \nabla \cdot \mathbf{q} \end{cases}$$

and the two equations of state $p = p(\rho, T)$ and $e = e(\rho, T)$. The heat flux can be written in terms of the temperature using Fourier's law (Eq.2.36). In order to solve the problem some *constitutive equations* must be added. Constitutive equations give a relation between the stress tensor (σ), the pressure (p) and the velocity (v). For a fluid in motion the stress tensor $\sigma = \sigma(p, \tau)$ is decomposed into:

$$\sigma = -p\mathbf{1} + \tau \tag{2.8}$$

where p is the *thermodynamic pressure*, which is a variable in terms of the density and the temperature, and τ is the *shear stress tensor.* According to the type of fluid the constitutive equations are:

- Ideals fluids: $\sigma = -p\mathbf{I}$
- Newtonian fluids: $\sigma = -p\mathbf{I} + \mathbf{C} : \nabla^s\mathbf{v}$
- Non-Newtonian fluids: $\sigma = -p\mathbf{I} + \mathbf{f}(\nabla^s\mathbf{v})$

For the Non-Newtonian fluids $\mathbf{f}$ is a non-linear function.

In the case of the Newtonian fluids there is a linear relation between the shear stress and the strain rate tensor $\nabla^s\mathbf{v}$. For the isotropic case, the shear stress tensor becomes:

$$\tau = \lambda tr(\nabla^s\mathbf{v})\mathbf{I} + 2\mu\nabla^s\mathbf{v} \tag{2.9}$$

Using this expression, the mean pressure is:

$$p - \bar{p} = K\nabla \cdot \mathbf{v} \tag{2.10}$$

with K $= \lambda + 2/3\mu$, which is the bulk viscosity. In the case of the incompressible fluid or when $\lambda = -2/3\mu$ (*Stokes assumption*), the thermodynamic pressure and the mean pressure are equal.

2.1.5 Boussinesq Approximation (Buoyancy)

Due to the fluid is influenced by change in temperature, the Boussinesq approximation must be taken into account. In Boussinesq approximation the density variations are considered and a linear dependence on temperature is registered as:

$$\rho = \rho_0 - \rho_0\beta\Delta T \tag{2.11}$$

where β is the thermal expansion and ΔT is the difference between the computed temperature and the room temperature. In stead of modify the value of the density, changes in gravity $\mathbf{g}$ are taken into account using the next approximation:

$$g_y = g_{y0} - g_{y0}\beta\Delta T \tag{2.12}$$

where g_{y0} is the initial value of the gravity in the y direction.

The value of the gravity will be update at each time step as

$$\mathbf{g}^* = (0, 1, 0)g_y$$

Applying the Boussinesq approximation the momentum equation (Eq.2.4) is:

$$\frac{\partial T}{\partial t} + \mathbf{v} \cdot \nabla \mathbf{v} = -\frac{1}{\rho}\nabla p + \nabla \cdot (\nu \nabla \mathbf{v}) - \mathbf{g}^* \tag{2.13}$$

2.1.6 Heat Equation

The heat equation equation describes the transport inside a physical system of a scalar quantity by convection and diffusion and the equation reads:

$$\rho c_p \frac{\partial T}{\partial t} + \rho c_p \mathbf{a} \cdot \nabla T - \nabla \cdot (k \nabla T) = q \tag{2.14}$$

where the scalar quantity T is the temperature, $\mathbf{a}$ is the convection velocity and q is the source term. It can be applied also for solids setting $\mathbf{a} = 0$ and the transport of the quantity be possible just by diffusion.

2.2 Stabilization Techniques

It is well known that for the heat equation (Eq. 2.14), for a convective dominant problem, strongly oscillatory values are produced . In order to avoid this behaviour a sufficiently small element size can be chosen, but the computational cost would be extremely expensive. Several technique have been proposed which add an extra term over the element interiors to the Galerkin weak form. This extra term involves the residual of the differential equation which is:

$$\mathscr{R}(T) = q - \underbrace{(\rho c_p \frac{\partial T}{\partial t} + \rho c_p \mathbf{a} \cdot \nabla T - \nabla \cdot (k \nabla T))}_{\mathscr{L}(T)} \tag{2.15}$$

where $\mathscr{L}(T)$ is the differential operator associated with the differential equation. The standard Galerkin finite element method applied to the heat equation and adding the stabilization term gives back the next equation in weak form:

$$\begin{aligned} \int_\Omega \rho c_p w \frac{\partial T}{\partial t} + \int_\Omega \rho c_p w \mathbf{a} \cdot \nabla T - \int_\Omega \nabla w \cdot k \nabla T \\ + \sum_e \int_{\Omega^e} P(w) \tau \mathscr{R}(T) d\Omega = \int_\Omega wq + \int_\Gamma w(\mathbf{n} \cdot \nabla T) d\Gamma \end{aligned} \tag{2.16}$$

where the stabilization term is:

$$P(w)\tau\mathscr{R}(T) \tag{2.17}$$

and $P(w)$ is an operator which depends of the stabilization technique used and τ is called the stabilization parameter.

Several technique (see [7]) are available as *Streamline-Upwind Petrov—Galerkin (SUPG)*, *Galerkin/Least-squares (GLS)* and *Algebraic Subgrid Scale Method (ASGS)*, which is a model of the *variational multiscale method (VMS)* techniques.

2.3 Variational Multiscale Methods

The VMS method provides the explanation of the origins of the stabilization parameters in residual based methods [14].

The VMS method basic idea is that the solution, u, can be decomposed into a *coarse scale* component,u_h, which can be solved by finite element method, and a *fine scale* solution, $\tilde{u}$, solved in a sub-grid scale. It is important to remark that the fine scale solution $\tilde{u}$ represents the error u - u_h of the coarse scale component [7].

VMS method exposition can be illustrated considering the heat equation (Eq. 2.14), with Dirichlet boundary conditions. The variational form of this boundary value problem, where $\mathscr{L}$ is a differential operator on the domain Ω (with boundary Γ), is:

$$\mathscr{L}u = f \quad in \quad \Omega \tag{2.18a}$$

$$u = g \quad in \quad \Gamma \tag{2.18b}$$

The variational form is:

$$a(w, u) = (w, \mathscr{L}u) = (w, f) \tag{2.19}$$

where the inner product is $(*,*)$, w is the weight function and $a(w,u)$ is a bilinear form. In this case $\mathscr{L}$ represents the heat equation.

In the VMS method the solution is assumed to be $u = u_h + \tilde{u}$ and the weight function $w = w_h + \tilde{w}$. The coarse and the sub grid quantities have the follow properties:

$$u_h = g \quad in \quad \Omega \tag{2.20a}$$

$$\tilde{u} = 0 \quad in \quad \Gamma \tag{2.20b}$$

Substituting u and w in the variational formulation:

$$a(w_h + \tilde{w}, u_h + \tilde{u}) = (w_h + \tilde{w}, \mathscr{L}(u_h + \tilde{u})) = (w_h + \tilde{w}, f) \tag{2.21}$$

the problem can be reformulated for the *coarse scale* (Eq. 2.22.a) and the *fine scale* (Eq. 2.22.b), where the first problem governs the resolved scales and the second one governs the unresolvable scales:

$$(w_h, \mathscr{L}u_h) + (w_h, \mathscr{L}(\tilde{u})) = (w_h, f) \tag{2.22a}$$

$$(\tilde{w}, \mathscr{L}\tilde{u}) + (\tilde{w}, \mathscr{L}(u_h)) = (\tilde{u}, f) \tag{2.22b}$$

Also the coarse scale and the fine scale can be rewritten using the adjoint operator $\mathscr{L}^*$ and the residual of the coarse scale $\mathscr{R}_{u_h}$ as:

$$(w_h, \mathscr{L}u_h) + (\mathscr{L}^* w_h, \tilde{u}) = (w_h, f) \tag{2.23a}$$

$$(\tilde{w}, \mathscr{L}\tilde{u}) = (\tilde{u}, f) - (\tilde{w}, \mathscr{L}(u_h) = (\tilde{w}, \mathscr{R}_{u_h}) \tag{2.23b}$$

The VMS method key point is to derive an analytic solution for the fine scale in terms of the coarse scale. Solving 2.23.b and substituting into 2.23.a single coarse scale equation with a proper accounting for the fine scale phenomena is provided [14].

Using *Green's function* the fine scale can be rewritten as:

$$\tilde{u} = -\int g'(\mathscr{L}u_h - f)d\Omega = -M'\mathscr{R}_{u_h} = -M'(\mathscr{L}u_h - f) \tag{2.24}$$

In this case g' i s the Green's function associated with the fine scale adjoint and finally the coarse scale becomes:

$$(w_h, \mathscr{L}u_h) - (\mathscr{L}^* w_h, -M'(\mathscr{L}u_h - f)) = (w_h, f) \tag{2.25}$$

where than the equation have to be solved for the coarse scale.

2.3.1 ASGS stabilization

In the current work the ASGS stabilization technique has been used and the derivation is obtained next.

ASGS stabilization technique has been introduced by Hughes and it is an efficient technique able to stabilize the convective term in a consistent manner. The idea in order to control the instability, is considering the unknown $T = t_h + \tilde{T}$ using VMS method. An important assumption of the ASGS method is that the unresolvable scale $\tilde{T}$ are forced to vanish on the element boundaries [1].

The simplest way to approximate the fine scale component $\tilde{T}$ is to assume:

$$\tilde{T} = \tau\mathscr{R}(T) \tag{2.26}$$

where τ is the stabilization parameter and $\mathscr{R}$(T) is the residual.

Considering the convective term following the assumption $T = t_h + \tilde{T}$ as:

$$\int_\Omega \rho c_p w\mathbf{a} \cdot \nabla T d\Omega = \int_\Omega \rho c_p w\mathbf{a} \cdot \nabla T_h d\Omega + \int_\Omega \rho c_p w\mathbf{a} \cdot \nabla \tilde{T} d\Omega \tag{2.27}$$

Expressing the fine scale convective term in the form:

$$\mathbf{a} \cdot \nabla \tilde{T} = \nabla \cdot (\tilde{T}\mathbf{a}) - \tilde{T} \nabla \cdot \mathbf{a} \tag{2.28}$$

and after using the divergence theorem, it can be rewritten as:

$$\int_\Omega \rho c_p w \mathbf{a} \cdot \nabla \tilde{T} d\Omega = \cancelto{0}{\int_\Gamma \rho c_p w \tilde{T} \mathbf{a} \cdot \mathbf{n} d\Gamma} - \int_\Omega \rho c_p \mathbf{a} \cdot \nabla w \tilde{T} d\Omega \tag{2.29}$$

where the term evaluated at the boundary vanishes because we imposed $\tilde{T} = 0$ on Γ.

Considering:

$$P(w) = -\rho c_p \mathbf{a} \cdot \nabla w \tag{2.30}$$

the stabilization term element by element is:

$$ASGS : -\sum \int_\Omega \rho c_p \mathbf{a} \cdot \nabla w \tau \mathscr{R}(T) d\Omega \tag{2.31}$$

The stabilization parameter τ has been taken following Codina's (2000) definition:

$$\tau = \left(\frac{\rho c_p}{\triangle t} + \frac{c_1 \rho c_p \|\mathbf{a}\|}{\triangle h} + \frac{c_2 k}{\triangle h^2} \right)^{-1} \tag{2.32}$$

where $\triangle t$ is the time increment, $\triangle h$ is the element size, $\|\mathbf{a}\|$ is the convective velocity norm, c_1 and c_2 are two constant parameters which typically in fluid mechanic take the value of $c_1 = 2$ and $c_2 = 4$.

The discrete form of the PDE with ASGS stabilization for **linear elements** will be:

$$\begin{aligned} &\int_\Omega \rho c_p w \frac{\partial T}{\partial t} + \int_\Omega \rho c_p w \mathbf{a} \cdot \nabla T - \int_\Omega \nabla w \cdot k \nabla T \\ &+ \sum \int_\Omega \rho c_p \mathbf{a} \cdot \nabla w \tau \left[\rho c_p \frac{\partial T}{\partial t} + \rho c_p \mathbf{a} \cdot \nabla T - q \right] = \int_\Omega w q + \int_\Gamma w (\mathbf{n} \cdot \nabla T) d\Gamma \end{aligned} \tag{2.33}$$

Applying $T_{approx} = \sum \mathrm{N}_j T_j$ and w $= N_i$ we get:

$$\begin{aligned}\int_\Omega \rho c_p N_i N_j \frac{\partial T_j}{\partial t} + \int_\Omega \rho c_p N_i (\mathbf{a}\cdot\nabla) N_j T_j - \int_\Omega \nabla N_i \cdot (k \nabla N_j T_j) \\ + \sum \int_\Omega \rho c_p \mathbf{a}\cdot\nabla N_i \tau \left[\rho c_p \frac{\partial N_j T_j}{\partial t} + \rho c_p (\mathbf{a}\cdot\nabla) N_j T_j - q \right] \\ = \int_\Omega N_i q + \int_\Gamma N_i (\mathbf{n}\cdot\nabla N_j) T_j d\Gamma\end{aligned} \tag{2.34}$$

Algebraic Problem

Converting the discrete heat equation problem with ASGS stabilization technique in algebraic form, Eq. 2.34 will be:

$$[\mathbf{M}+\mathbf{N}]\partial_t T + [\mathbf{C}-\mathbf{K}+\mathbf{S}+\mathbf{B}_{out}]T = \mathbf{q} \tag{2.35}$$

where,

$\mathbf{M} = \int_\Omega \rho c_p N_i N_j d\Omega$: Mass matrix

$\mathbf{N} = \int_\Omega (\rho c_p \mathbf{a}\cdot\nabla N_i)\tau(\rho c_p N_j) d\Omega$: Mass matrix fine scale

$\mathbf{C} = \int_\Omega \rho c_p N_i (\mathbf{a}\cdot\nabla) N_j d\Omega$: Convective matrix

$\mathbf{K} = \int_\Omega \nabla N_i \cdot (k\nabla N_j) d\Omega$: Diffusion matrix

$\mathbf{S} = \int_\Omega (\rho c_p \mathbf{a}\cdot\nabla N_i)\tau(\rho c_p \mathbf{a}\cdot\nabla N_j) d\Omega$: Stabilization term matrix

$\mathbf{B}_{out} = \int_\Gamma N_j N_i (\mathbf{a}\cdot\mathbf{n}) d\Gamma$: Outflow boundary matrix

$\mathbf{q} = \int_\Omega N_i q + \tau(\rho c_p \mathbf{a}\cdot\nabla N_i) q d\Omega$: Load vector

2.4 Thermodynamics and Heat Transfer

The *heat* is the form of energy which can be transferred from a system to another due to a temperature difference and the *rate* of this energy transfer is called *heat transfer*.

The heat transfer can be expressed as the science which deals with the transfer of the thermal energy from a point to another within a medium or from one medium to another due to the occurrence of the a temperature difference [14].

Diverging from a pure thermodynamic study, which gives the amount of the heat transfer between two different equilibrium states, an heat transfer study provides the indication about *how long* the process will take, which is more interesting in a practical point of view, and it also deals with *non-equilibrium* phenomena, where a lack of thermal equilibrium is present. *First law* of thermodynamics, which states that the increment energy transfer rate to a system has to be equal to the increment energy transfer rate of that system, and *second law* of thermodynamics, which requires that heat is moving in the direction of decreasing temperature, put the basis of the heat transfer science [11]. It is important to underline that the temperature difference is the cause of the heat transfer and the heat transfer direction depends of the temperature gradient.

The heat can be transferred in three different way: *conduction, convection* and *radiaton* (in the present documents only the conduction and the convection will be taken in consideration). For all of them, in order to achieve the heat transfer, a temperature difference is needed, and for each of them it occurs from the medium with high temperature to the medium with low temperature.

2.4.1 Conduction

From a microscopic point of view, conduction can be expressed as the energy transfer from a medium where particles have more energetic level to a different medium, where the particles are less energetic.

Heat is transported in solid materials by both lattice vibration waves (phonons) and free electrons [6].

The rate of heat conduction through a medium depend of several factors as the material property, the temperature difference and the geometry. *Fourier's law*, the law of heat conduction, states:

$$\dot{Q}_{cond} = -kA\nabla T \quad [W] \tag{2.36}$$

where k is the *thermal conductivity* of the material, which describes the material capacity to conduct heat, A is the surface and ∇T is the temperature gradient.

The Fourier's law indicates that the rate of the heat conduction is proportional to gradient temperature and the heat is conducted toward the low temperature.

Thermal conductivity

The property that characterizes the ability of a material to transfer heat is the thermal conductivity k. The thermal conductivity, according to the medium, can vary in a wide range of intervals. Big thermal conductivity values are associated with good heat conduction, which is typical of metals and pure crystals. Meanwhile small thermal conductivity values are related with gases and insulating material: in this case the material are called *insulators* [6].

The thermal conductivity is a tensor which contains the thermal properties of the material and its units are [W/(mK)].

Thermal diffusivity

An important material property is the thermal diffusivity, which deals with how fast the heat is diffused through a material and it gives the reason between the heat conduction and the heat stored in a material. It is defined as:

$$\alpha = \frac{k}{\rho c_p} \quad \left[\frac{m^2}{s}\right] \tag{2.37}$$

where ρc_p is the *heat capacity* and it gives the amount of the energy stored per unit of volume. The energy required in order to rise up the temperature of a unit mass of a substance by one degree at constant pressure is called *specific heat* c_p.

It is important to point out that as the thermal diffusivity is higher the heat propagation is faster.

2.4.2 Convection

The phenomena which is associated with an energy transfer between a solid and the adjacent fluid in motion (liquid or gas) is well known as convection.

Conduction and convection are related due to both mechanisms need a physical medium in order to be effective. Heat transfer in solid occurs only by conduction meanwhile in liquid, heat transfer can happen by conduction or convection. Convection is related with a bulk fluid motion and conduction in absence of it.

Convection involves a combination between the effects of conduction and fluid motion. The heat transfer between a static bulk fluid next to a solid surface is related only by the conduction (pure conduction), meanwhile, for a bulk fluid in movement, as the motion is faster, greater is the convection.

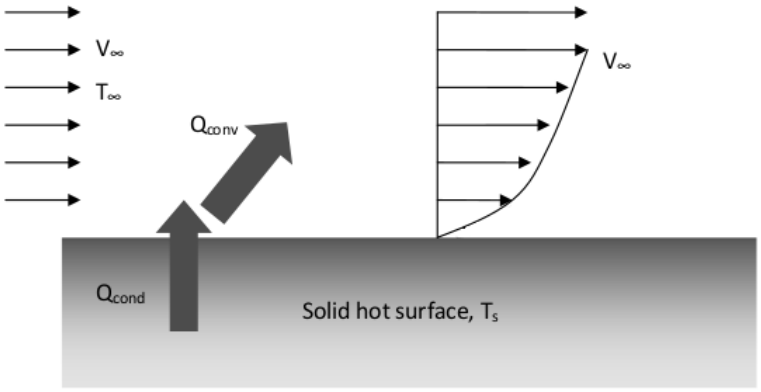


Figure 2.1: Convection and conduction between a solid and a fluid (Bahrami).

A general description of the convection includes concepts as *laminar and turbulent flow* and *velocity and thermal boundary layer*. Also dimensionless *Reynold, Prandtl* and *Nusselt numbers* concepts will be introduced and finally a relation between heat and momentum transfer in turbulent flow is described.

Considering a contact between a hot solid surface and a cool bulk fluid air, an important concept of energy exchange between them must be considered: heat is first transferred by conduction because the velocity of the fluid's particles is zero near the wall, and than this energy is taken away from the solid surface by convection (Figure 2.1).

According if the movement of the fluid is forced, the convection is called forced convection; in contrast convection is called natural if the fluid motion is caused by buoyancy forces that are induced by density differences due to the temperature variation [11].

It has been proved the convection is proportional to the difference temperature and this dependence is described by *Newton's cooling law*.

The rate of convection heat transfer is expressed as:

$$\dot{Q}_{conv} = hA_s(T_s - T_\infty) \quad [W] \tag{2.38}$$

where h is the *convection heat transfer coefficient* in $W/(m^{2\circ}C)$, A_s is the surface area between the fluid and the solid, T_s is the temperature of the solid surfaces, and T_∞ is the fluid temperature far away from the surface.

The convection heat transfer coefficient value is experimental determinate and depends of the nature of the fluid motion, the bulk fluid velocity, the properties of the fluid and surface geometry.

Due to the presence of both fluid motion and heat transfer, convection is a really complex mechanism. It is known that convection increases heat transfer and the rate of heat transfer enhances as the velocity is higher.

Convection's complexity can be understand , and also experimentally proved, by virtue of the dependence of many variable as *cinematic viscosity* ν*, conductivity* k*, specific heat* c_p*, fluid velocity, geometry, roughness* of the solid surface and to the *fluid regime flow* [11].

It is important to introduce a concept known as *no-temperature-jump-condition* which occurs in heat transfer: when two medium with different temperature enter in contact, heat transfer takes place until both bodies reach the same temperature at the point contact.

The no-temperature-jump-condition associated with a no-slip condition imply that the heat transfer between solid and fluid, at the point contact, is through *pure conduction*, and it can be expressed as:

$$\dot{q}_{conv} = \dot{q}_{cond} = -k_{fluid}\frac{\partial T}{\partial y}|_{y=0} \quad \left[\frac{W}{m^2}\right] \tag{2.39}$$

where T represents the temperature distribution in the fluid and $(\partial T/\partial y)_{y=0}$ is the temperature gradient at the surface.

The *convection heat transfer coefficient* can be obtained equating Eq.2.38 and Eq.2.39 as:

$$h = \frac{-k_{fluid}(\partial T/\partial y)_{y=0}}{T_s - T_\infty} \quad \left[\frac{W}{m^{2\circ}C}\right] \tag{2.40}$$

2.4.3 Nusselt Number

Nusselt number, viewed also as the *dimensionless convection heat transfer coefficient*, represents the enhancement of heat transfer through a fluid layer as a result of convection relative to conduction across the same fluid layer [11].

The Nusselt number is physically expressed as the ratio between the heat transfer by convection and the heat transfer by conduction, both for a fluid.

Their ratio gives:

$$\frac{\dot{q}_{conv}}{\dot{q}_{cond}} = \frac{hL}{k} = Nu \tag{2.41}$$

The Nusselt number gives the convection's efficiency and as the number is larger, than the more effective is the convection. Pure conduction corresponds at a $Nu = 1$.

Variation of the Local Nusselt Number

Nusselt number is not locally constant and it can vary due to the geometry and the fluid regime flow. An example of the locally Nusselt number variation can be explained studying the flow around a cylinder for different regimes.

Flows around cylinder carry difficulties due to a phenomena called *flow separation* and it occurs when:

- an inflection point exists in the velocity profile because the velocity is zero at the surface due to the non slip condition
- an adverse pressure gradient occurs in the direction flow

When the pressure gradient change sign and from favorable ($\partial p/\partial x < 0$) becomes adverse ($\partial p/\partial x > 0$), the boundary layer flow slows down, it gets thicker and a inflection point appears on the profile velocity. As a consequence fluid particles are transported away from the surface and the stream line emerges at the separation point (Figure 2.2). At some point the flow also starts to change the direction near the surface (flow reversal) with a consequence change of sign of the shear stresses from negative to positive, and they reduce to zero at the separation point.

$$\left(\frac{\partial u}{\partial y}\right)_{wall} = 0 \quad at\ separation\ point$$

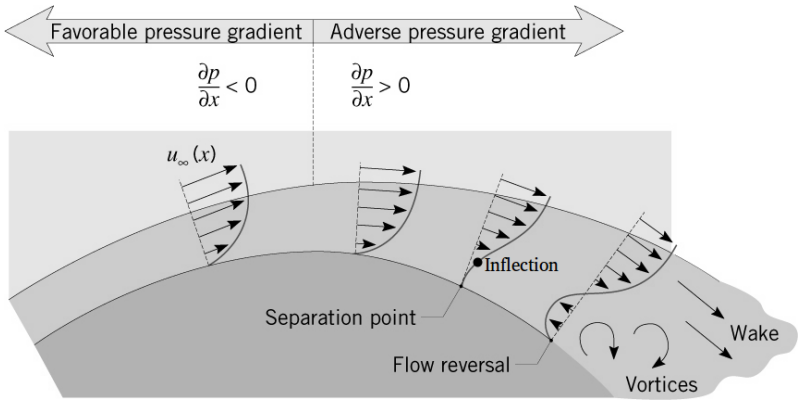


Figure 2.2: Velocity profile depending on the adverse pressure gradient and separation point. (Baker)

Depending on the flow regime, the separation occurs at different points at the cylinder surface. Figure 2.3.a shows that the separation it registered at $\theta \approx 80$ for a laminar flow, meanwhile for a turbulent regime, the separation occurs at $\theta \approx 140$. In the case of the laminar flow the separation occurs before due to this type of regime is really weak to the adverse gradient pressure whereas the turbulent flow is more resistant and that is why the separation occurs later.

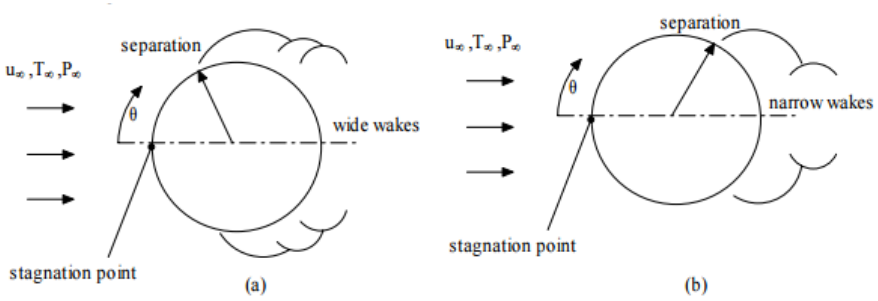


Figure 2.3: Separation flow over a cylinder(MPGC).

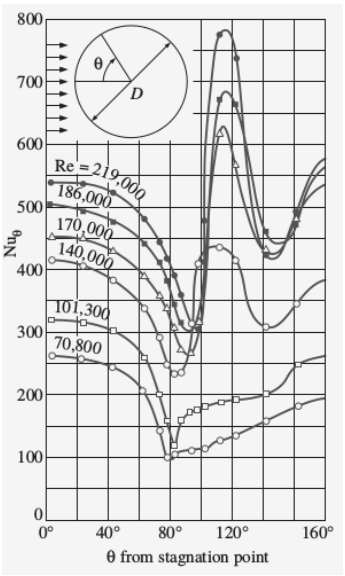


Figure 2.4: Nu vs θ (Cengel)

Due to the flow separation the heat transfer is influenced and consequently the Nusselt number variates around the surface of the cylinder in cross flow air (Figure 2.4). As the flow reaches the cylinder, the Nu value is relatively high and it is registered at the stagnation point ($\theta = 0$). Nu decreases with increasing of θ, as a result of the thickening of the laminar boundary layer, reaching the minimum at $\theta \approx 80$ (separation in laminar flow). After that, Nu increases really sharp with θ because of the transition between laminar to turbulent regime, and than it decreases until the flow separation point in turbulent flow is reached ($\theta \approx 140$).

The average Nusselt number for a flow across cylinders can be computed in a compact form using the next formula:

$$Nu_{cyl} = \frac{hD}{k} = CRe^m Pr^n \tag{2.42}$$

which, n = 1/3, and the constant C and m depend on the type of the shape and the regime flow.

All the fluid properties are evaluated at the film temperature , which is defined as $T_f = (T_s + T_\infty)/2$. Film temperature is the average between the surface temperature and the free stream temperature.

According the Zukauskas's correlation, for different range of Reynolds number, the constant values can be obtained using next table

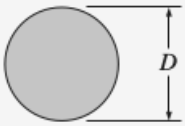
Cross-section of the cylinder	Fluid	Range of Re	Nusselt number
Circle 	Gas or liquid	0.4–4 4–40 40–4000 4000–40,000 40,000–400,000	$Nu = 0.989Re^{0.330}\ Pr^{1/3}$ $Nu = 0.911Re^{0.385}\ Pr^{1/3}$ $Nu = 0.683Re^{0.466}\ Pr^{1/3}$ $Nu = 0.193Re^{0.618}\ Pr^{1/3}$ $Nu = 0.027Re^{0.805}\ Pr^{1/3}$

Figure 2.5: Correlation for different Nu number for a flow over a cylinder for different Re numbers (Cengel).

2.5 Fluid Dynamics

The fluid dynamic is the discipline which describes the fluid flow and the effects at the boundary, where it can be a solid surface or another fluid. Due to the wide variety of fluid flows, fluids can be classified into different categories and also some fundamental fluid properties have been introduced. In this chapter also the fluid dynamic problem described by the *Navier-Stokes Equations* will be discussed.

2.5.1 Fluid Properties

Viscous versus Inviscid Flow

In order to understand the difference between *viscid and inviscid flow*, an important property called *viscosity* has to be introduced.

Viscosity is an internal resistance to flow. This property is the product of the internal friction between the different fluid layers, where in liquids it is generated by the cohesion forces between molecules meanwhile for gases it is generated due to molecular collision. Such common fluids follow a linear relation between shear and resulting strain rate. Also for the common linear fluids it can be demonstrated that the shear stresses are proportional velocity gradient and the proportional coefficient is called viscosity:

$$\tau = \mu \frac{\partial u}{\partial y} \quad \left[\frac{N}{m^2}\right] \tag{2.43}$$

Newtonian fluids are the linear fluids which follow Eq.2.43, meanwhile fluids which do not follow this relation are defined as *nonewtonians*. A fluid which has no resistance to shear stresses ($\mu = 0$) is called *inviscid* or *ideal*.

Viscosity can be expressed also divided by ρ, where the mass unit is canceled, and it is called *kinematic viscosity*

$$\nu = \frac{\mu}{\rho} \quad \left[\frac{m^2}{s}\right] \tag{2.44}$$

For practical calculation and under certain conditions, fluids like water and air can be assumed as Newtonian fluids.

It is important to underline that temperature has strong effects on viscosity. For liquids viscosity decreases with temperature increasing and on the other end, for gases viscosity increases with temperature.

Compressible versus Incompressible Flow

The density variation of the fluid characterizes the flow classification as *compressible or incompressible.* Generally flows in which the density is constant in an infinitesimal volume that moves with the flow velocity are considered incompressible. For the flows where the density variations can not be neglected, are considered compressible.

An important indicator of the compressibility is the Mach number, which is defined as:

$$M_\infty = \frac{V_\infty}{a_\infty} \tag{2.45}$$

where V_∞ is the free stream flow and a_∞ is the speed of sound (in dry air, at 20°C, a = 1230 km/h).

Theory and observations in wind tunnels suggest that most flows may be treated as incompressible (constant density) until the Mach number is sufficiently high. For M < 0.3 a flow can be considered incompressible.

Reynolds Number

Reynold number is an important dimensionless number which determine the behaviour of a fluid flow. It is defines as:

$$Re = \frac{\rho u D}{\mu} \tag{2.46}$$

Where u is the velocity, D is characteristic linear dimension of the problem, and μ is the dynamic viscosity.

The three different flow regimes can be quantified according to Re as

- Re $<$ 2300 laminar flow
- 2300 $\leq$ Re $\leq$ 10.000 transitional flow
- Re $>$ 10.000 turbulent flow

For a circular pipeline, the characteristic linear dimension corresponds with the diameter of the pipe, meanwhile for a sections such as squares, rectangular or annular ducts where the height and width are comparable, characteristic linear dimension is defined as:

$$D_h = 4\frac{A_c}{pr} \tag{2.47}$$

where A_c is the section of the pipe and pr is the wet perimeter.

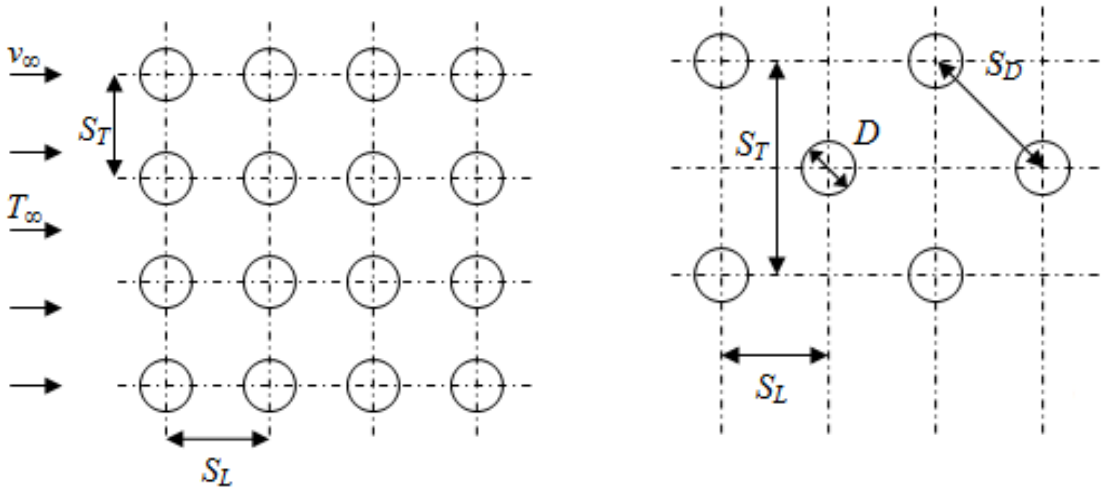
The Reynolds number can be also defined as:

$$Re = \frac{F_i}{F_v} \tag{2.48}$$

where F_i represents the inertial forces and F_v the viscous forces. Inertial forces are defined due to the momentum of the fluid. This is usually expressed in the momentum equation by the term $(\rho u)u$. So the momentum (inertia) is higher with the increasing of the velocity and it also depends to the density. As in classical mechanics, a force that can counteract or counterbalance this inertial force is the force of friction or viscous forces. In fact Reynolds number is an important viscous effect indicator: due to it is inversely proportional to the viscosity, for a large Reynolds number, the viscous effects do not play an important role about the flow behaviour. On the other end, for a dominant viscous effect, Reynolds number is small and it indicates loss of energy due to the viscous forces.

2.5.2 Flow Across Staggered Tube Banks

Flow across bank of tubes is a really common case of study in thermo-fluid dynamic. The two most common geometric arrangements of a tube bank are shown in Figure 2.6.



((a)) Aligned configuration. ((b)) Staggered configuration.

Figure 2.6: Tube banks (NPTEL).

For each configuration, D is the diameter of tube, S_L is the longitudinal spacing, and S_T is the transverse tube spacing. For the staggered configuration S_D corresponds to the diagonal spacing.

In order to determine the maximum velocity the continuity equation for steady incompressible flow is required.

For the *aligned configuration* the maximum velocity occurs at the minimum flow area and the continuity equation can expressed as $\rho v_\infty S_T = \rho v_{max}(S_T - D)$. The maximum velocity becomes:

$$v_{max} = \frac{S_T}{S_T - D} v_\infty \tag{2.49}$$

In the case of the *staggered configuration* the maximum velocity occurs at the diagonal cross section and the maximum velocity is:

$$v_{max} = \frac{S_T}{2(S_T - D)} v_\infty \tag{2.50}$$

It has been proved experimentally that the flow behaviour in the first row of tubes is really similar to a flow around a single tube. When the flow reaches a tube in the second and subsequent rows the behaviour is very different because of the wakes and the turbulence formed after the upstream tubes. The consequence is an increasing of the heat transfer coefficient and the level of the turbulence with the rows number. It also be proved the the level of turbulence and the heat transfer coefficient remain constant after the few first rows [11].

2.5.3 Peclet Number

Another dimensionless number really important in transport phenomena is the Peclet number (Pe). Peclet number characterizes the relative importance of convective and diffusive effects in a given flow and it expresses the ratio of convective to diffusive transport.

For 1D case, Peclet number is:

$$Pe = \frac{uL}{k} \tag{2.51}$$

where L is the characteristic length, u the local flow velocity and k is the thermal conductivity.

It has been proved that when the convection is dominant corresponds to Peclet values greater than one and, due to the effects of the convection, instabilities in the numerical solution are generated.

Laminar versus Turbulent Flow

In *Laminar flow* the motion of the particles of fluid is characterized by smooth streamlines and highly-ordered motion. As the flow experiments sudden fluctuations of the velocity field, pressure filed and temperature filed and the particle paths become completely irregular, the flow is defined *turbulent*.

Steady versus Unsteady (Transient) Flow

The *steady state* flow implies that the fluid does not experiment any properties change in a point domain with time. Basically the fluid properties can be different from point to point in the domain, but at any fix point they must be constant with time. Energy, volume and mass do not vary for a steady state case. *Unsteady flow* is characterized by the conditions at one point vary with time.

2.5.4 Velocity Boundary Layer

The velocity boundary layer is a concept introduced by Ludwig Prandtl and its main assumption is that the viscous forces are negligible everywhere, except near the solid boundaries because the no-slip condition.

The most important assumption is that the thickness layer is extremely thin compared to the other length scale due to the no-slip condition and the fluid layer velocity adjacent to the surface body becomes zero. The consequence results is a slow down of the fluid neighboring layers due the particles friction because of the adjoining layers different velocity. The result is the boundary layer development

where, from the solid surface until a certain distance δ the velocity in the normal direction describes a parabolic profile. In the boundary layer the fluid velocity tangential to the solid surface goes from zero at y=0 to nearly u_∞ at $y = \delta$. The boundary layer thickness, δ, can be considered as the distance y=0 from the solid surface at which $u = 0.99u_\infty$ [11]. With the velocity boundary layer generation the flow can be divided into two regions: the velocity boundary region and the inviscid flow region, where in the last one the velocity remains constant and the viscous effects are neglected.

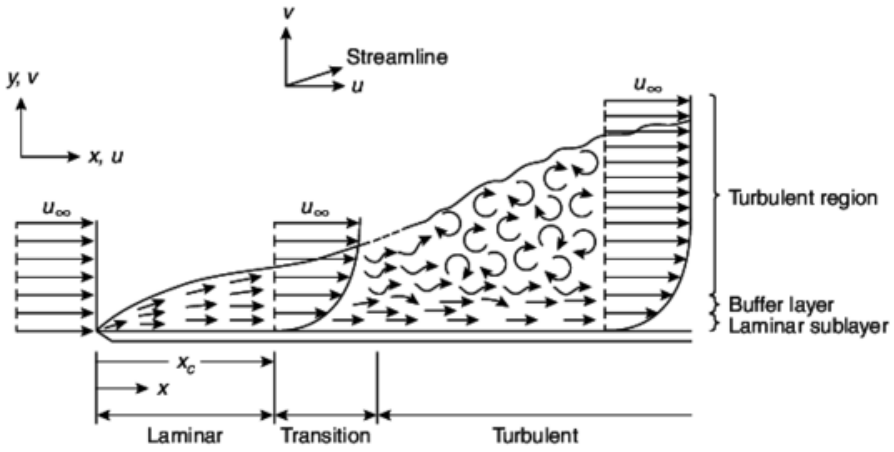


Figure 2.7: Velocity boundary layer generation for a different fluid regime (Atreya).

In the turbulent regime boundary layer three different sub layers can be recognized (Figure 2.7). Close to the wall, a really thin layer called *laminar sublayer* is generated and it is characterized to have viscous dominant effects. Next to the laminar layer appears the *buffer sublayer*, where, even if the turbulent effects are important, the flow is dominated by the viscous effects. Finally above the buffer sublayer, where the turbulent effects are dominant, a large layer called *turbulent region* is developed.

No-Slip Condition

No-slip condition states that due to the viscous forces the fluid flow velocity is equal to zero at the contact surface with a solid. This condition ensures that, excluding rarefied gases, then all fluids at a point of contact with a solid take on the velocity of the surface:

$$\mathbf{v}_{fluid} = \mathbf{v}_{solid} = 0 \tag{2.52}$$

2.5.5 Thermal Boundary Layer

The thermal boundary layer develops when a fluid with a certain temperature T_∞ flows is in contact with a solid surface at a different temperate T_s. As a consequence a thermal equilibrium will be achieved through the contact between the fluid particles temperature with the solid particles. As a result due to the energy absorption between solid-fluid particles, a temperature profile is generated with the normal direction to the surface (Figure 2.8). The region where the temperature variation is important is called thermal boundary layer. The thickness δ can be defined as the distance from the solid surface where the difference temperature $(T - T_s)$ is equal to $0.99(T - T_s)$.

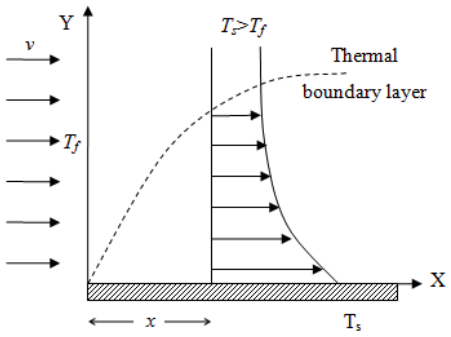


Figure 2.8: Boundary layer for a fluid temperature greater than the solid surface temperature (NPTEL).

2.5.6 Prandtl Number

The dimensionless Prandtl number is a ratio of two molecular transport properties and it describes the relative thickness between the velocity boundary layer and the thermal boundary layer.

The Prandtl number is defined as:

$$Pr = \frac{\mu c_p}{k} \tag{2.53}$$

The physical meaning from a molecular point of view of the Prandtl number can be explained as the relation between the momentum molecular diffusivity and the heat molecular diffusivity. For a Prandtl number nearly close to 1, common for gases, both the momentum and the heat dissipate through the fluid at about the same rate [11]. For the case where $Pr < 1$, the thermal boundary layer δ_t is thicker than the velocity boundary layer δ due to the heat diffuse quickly. It is typically of

liquid metals behaviour where the conductivity value is very high and the thermal diffusivity dominates. From the other hand heat diffuses very slow compared with the momentum for $Pr > 1$, where in this case the thermal boundary layer is thinner (Figure 2.9). It is associated for oils behaviour, where the momentum diffusivity dominates.

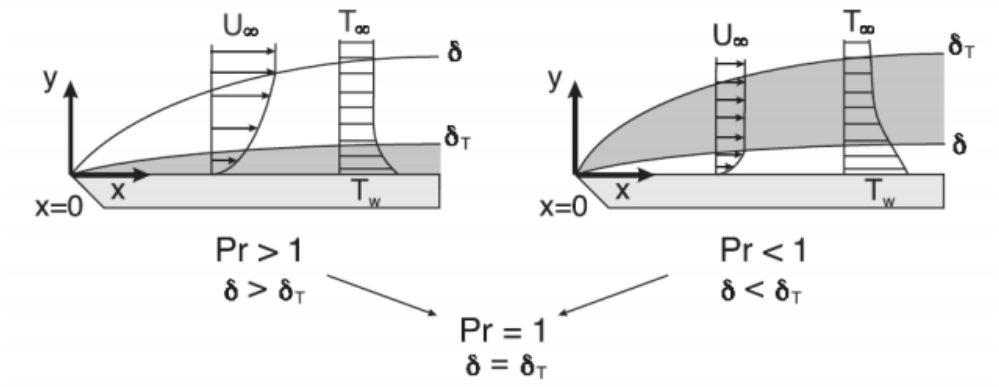


Figure 2.9: Thermal boundary layer and velocity boundary layer comparison for different Prandtl numbers (Bahrami).

2.5.7 Navier-Stokes Equations

The fluid dynamic problem for a *Newtonian fluid* is governed by the *Navier-Stokes Equations*, where the dynamic effect due to the external and internal forces over a fluid are stated.

For a transient flow of an incompressible viscous flow, the problem is governed by the Navier Stokes equations, which are the *mass conservation equation* (Eq. 2.1) plus the incompressibility condition (Eq. 2.2) and the *momentum equation* (Eq. 2.4):

$$\rho \left[\frac{\partial \mathbf{v}}{\partial t} + (\mathbf{v} \cdot \nabla)\mathbf{v} \right] = \nabla \cdot \sigma + \rho \mathbf{b}$$

$$\nabla \cdot \mathbf{v} = 0$$

Under the incompressibility condition , which states that the density remains constant during the motion, the momentum equation can be rewritten as

$$\frac{\partial \mathbf{v}}{\partial t} + (\mathbf{v} \cdot \nabla)\mathbf{v} - \nu \nabla^2 \mathbf{v} + \nabla p = \mathbf{b}$$

The Navier-Stokes Equations carry some issues which makes the incompressible flow problems difficult. The first difficulty is related with the presence of the convective term in the momentum equation. This difficulty arises with instabilities for the convective dominant case for the standard Galerkin formulation. Such difficulties can be overcome using the stabilization technique discussed in chapter 2.2.

The second difficulty is associated with the constrain on the velocity field, which must be divergence free [7]. Due to that pressure is treated as another degree of freedom (DoF) needed to satisfy the incompressibility constrain and originating a coupling between pressure and velocity unknowns.

For an incompressible flow an appropriate combination of element interpolation function for the velocity and pressure is needed and the solvability depend on the *Ladyzenskaja-Babuska-Brezzi* (LBB) condition.

2.5.8 Turbulence

Turbulence is a highly complex physical phenomenon that is pervasive in flow problems of scientific and engineering concern [14]. Due to its complexity it is one of the unsolved problems in science and it entails theoretical difficulties and high numerical computation demand. A definition of turbulence can be expressed as an irregular flow condition showing random variations with respect to both time and space coordinates with discernible statistical properties [14]. Even if it is difficult to define a standard definition of this phenomenon, several general characteristics are recognizable. For a turbulent flow the most important are:

- highly nonlinear
- fluctuations
- vorticity
- dissipation
- highly diffusive
- random process in time
- fully three-dimensional and fully time-dependent

A fluid becomes turbulent when the Reynold number Re exceeds a critical value and small perturbations can grow spontaneously and may equilibrating as finite amplitude disturbances. The non-linearity of turbulence is evident since it is the final state of a nonlinear transition process. The disturbances can increase and get more complicated, reaching an unrepeatable unpredictable state (turbulence) [5]. Turbulence involves irregular and chaotic fluctuations of the dependent-filed quantities

(pressure, temperature, velocity,...). Typically turbulence involves identifiable structures called *eddies*. These structures are recognizable as regions in the fluid where the flow rotates, deforms and divides. Eddies's size increase with the Reynolds number and the characteristic size of the largest eddies is the width of the turbulent region. In turbulence the chaotic vorticity is characterized to be three-dimensional where vortexes stretch and turn by themselves, so without a three dimensional vorticity the turbulence is not real.

From a engineering point of view turbulence can be simplified without taking into account the detailed resolution and considering the average flow description. In order to obtain an accurate study of a turbulent flow, it is extremely important to generate a mesh resolution respecting the same order of the smallest length scale. The smallest eddy in turbulent flow are $\approx \mathcal{O}(0.001L)$ where L is the characteristic flow dimension, and for a three dimension mesh the number of nodes are $\approx 10^9$ [Eddy]. It is important to underline that the computational cost for a relevant industrial problem with complex geometries is extremely expensive.

Classification of Turbulent Models

Even if in the present work any turbulent model has been used it is important to introduce them due to their importance in CFD.

Different turbulence models have been formulated and especially the next three are distinguished due to their importance.

- Direct Numerical Simulation (DNS)
- Large Eddy Simulation (LES)
- Reynolds Avaraged Navier-Stokes (RANS)

The DNS simulation is used for high Reynolds numbers in order to solve all the turbulence's structures. In order to capture all the turbulence behaviour, the time step and the mesh element size have to be really small and consequently the simulation is extremely expensive. The DNS computation required $\approx Re^{9/4}$ number of elements and the total computational time is proportional to $\approx Re^3$.

The LES simulation solves the very wide range of time and length scales ignoring the smallest length scales, which are solved computing a time and spatial averaging.

Finally, the RANS models compute the avarage flux and they approximate the turbulence effects over the avarage flux. The formulation consists in the decomposition of the velocity field (u) into the avarage flux (U) and the small oscillations (u') at each time step, adding new unknowns at the system, of equation and approximations are introduced near the walls.

$$u = U + u'$$

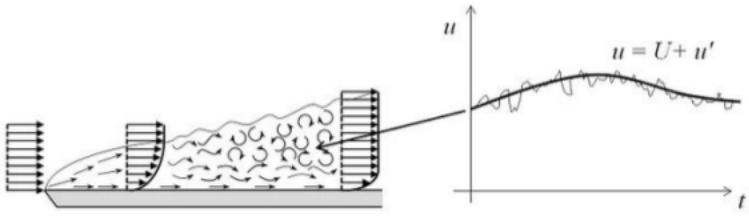


Figure 2.10: Velocity filed turbulent regime (Ferziger).

2.5.9 Heat and Momentum Transfer in Turbulent Flow

Important cases of study in engineering practice are related with turbulence flow, where due to the high and suddenly fluctuations of particles, the flow is characterized with a *eddies* formations in the boundary layer area.

In the laminar flow regime heat transfer and momentum are transferred along the stream line by molecular diffusion. In the case of turbulent regime, due to the eddies transversal motion, momentum and energy are transported to different fluid regions and, as a results, momentum and heat transfer are enforced. As a consequence also friction coefficient and heat transfer coefficient increase.

In turbulent regime all the properties as velocity, pressure, temperature and density (in compressible flow) can be decomposed in the *mean value* and the *fluctuating component*. So the properties can be expressed as

$$v = \bar{v} + v'$$

$$p = \bar{p} + p'$$

$$T = \bar{T} + T'$$

Even if the fluctuation values are usually a few percent of the mean value, the high frequency of eddies make them very effective for the transport of thermal energy and momentum. The eddies motion and diffusivities (which are zero at the wall) in the turbulent boundary layer imply larger temperature and larger velocity gradient for the turbulent case compared with the laminar case. Due to the no-sleep condition the eddies motion decrease near the wall and, as a result, the velocity and temperature profiles are are very steep in the thin layer adjacent at the wall surface and uniform in the core of the turbulent boundary region. The larger velocity and temperature gradient near the wall imply also large shear stresses and heat transfer rate.

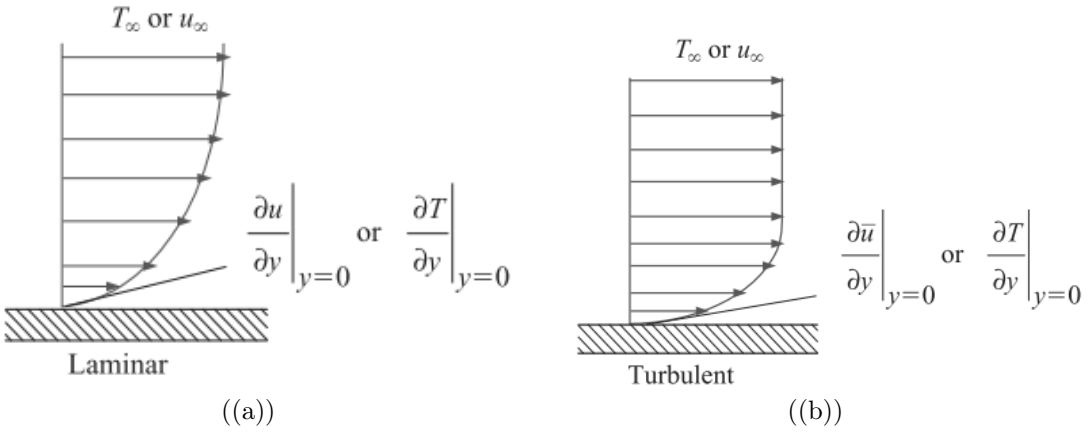


Figure 2.11: Velocity and temperature gradients at the wall for laminar and turbulent regime [11].

2.6 Finite Element Method

A problem described with partial differential equations, in order to be well posed, needs boundary conditions (BC), and, in the case of a transient problem, also initial conditions (IC).

The governing equations along with the boundary conditions define the *strong form* of the problem. The boundary conditions can be classified in three different type: *Dirichlet, Neumann* y *Robin*.

The Dirichlet BC prescribes the value of the unknown function on the boundary, the Neumann BC imposes the value of the normal gradient of the unknown function and the Robin BC defines a combination between the Dirichlet BC and Neumann BC.

Before to define the *weak form* of the problem the concept of the *weighting functions* has to be introduced.

The weighting functions belong to a function space where all the functions are square integrable, first derivatives square integrable over the computational domain and the functions vanish on the domain where the Dirichlet boundary conditions are imposed.

Mathematically this space can be described as:

$$\mathscr{H}^1 = \{v \in \mathscr{R} \backslash \int \|v\|^2 < \infty \quad | \int \|\nabla v\|^2 < \infty \quad | \quad v = 0 \quad on \quad \Gamma_D\} \equiv \mathscr{H}_0^1 \tag{2.54}$$

$\mathscr{H}^1$ is a class of Sobolev space where the functions and its derivatives are square integrable and bounded.

Two classes of functions have to be defined in order to define the weak form: the *weighting* functions and the *trial* functions.

The first class is defined by the space function $\mathscr{V}$ and it coincides with space introduced before,

$$\mathscr{V} = \mathscr{H}^1_{\Gamma_D}$$

The second class of functions, called as trial functions, has some similarities with the weighting functions except that these test functions have to satisfy the Dirichlet conditions on Γ_D. The second class is defined as:

$$\mathscr{S} = \{u \in \mathscr{H}^1 \quad | \quad u = u_D \quad on \quad \Gamma_D\} \equiv \mathscr{V} + \bar{u}_D \tag{2.55}$$

where $\bar{u}_D$ is the value of the function defined as a Dirichlet boundary condition.

In the finite element method, $\mathscr{V}$ and $\mathscr{S}$ are approximated to a finite subset defined as $\mathscr{V}^h$ and $\mathscr{S}^h$. They are characterized by a discretization of the computational domain Ω into elements domains.

$$\Omega = \bigcup_{e=1}^{n} \Omega^e \quad \text{and} \quad \Omega^e \cap \Omega^f \emptyset \quad \text{for} \quad e \neq f$$

where n is the number of elements.

The subset $\mathscr{V}^h$ and the subset $\mathscr{S}^h$ are defined as:

$$\mathscr{V}^h = \{w \in \mathscr{H}^1(\Omega) \quad w|_{\Omega^e} \in \mathscr{P}_m(\Omega^e) \quad \forall e \quad and \quad w = 0 \quad on \quad \Gamma_D\} \equiv \mathscr{H}^1_0 \tag{2.56}$$

$$\mathscr{S}^h = \{u \in \mathscr{H}^1(\Omega) \quad u|_{\Omega^e} \in \mathscr{P}_m(\Omega^e) \quad \forall e \quad and \quad u = u_D \quad on \quad \Gamma_D\} \equiv \mathscr{H}^1_0 \tag{2.57}$$

where $\mathscr{P}_m$ is the *finite element interpolation space.*

2.6.1 Weighted Residual Method

The *weighted residual method* (WRM) is a technique for solving partial differential equations with FEM and it consists to transform the governing equations in a equivalent integral expression. In order to obtain the equivalent integral form the governing equation has to be multiplied by an arbitrary *weighting function* w and integrate over the domain.

2.6.2 Weak and Algebraic Formulation of the Heat Equation

For the steady state heat equation posed in strong form as

$$\begin{cases} \mathbf{a} \cdot \nabla T - \nabla \cdot (\nu \nabla T) = q \quad in \quad \Omega \\ T = T_D \quad in \quad \Gamma_D \\ \nu \frac{\partial T}{\partial n} = T_N \quad in \quad \Gamma_N \end{cases}$$

and multiply the governing equation by a test function w

$$\int_\Omega \rho c_p w \mathbf{a} \cdot \nabla T d\Omega + \int_\Omega w \nabla \cdot (k \nabla T) d\Omega = \int_\Omega w q d\Omega \quad \forall w \ \in \mathscr{V}$$

Integrating by parts the second order term (diffusive term) we obtain the weak form of the problem which is:

$$\begin{aligned} &\int_\Omega \rho c_p w \mathbf{a} \cdot \nabla T d\Omega - \int_\Omega \nabla w \cdot k \nabla T d\Omega \\ &= \int_\Omega w q d\Omega + \int_\Gamma w (\mathbf{n} \cdot \nabla T) d\Gamma \qquad \forall w \ \in \mathscr{V} \end{aligned} \tag{2.58}$$

The previous equation shows that the the second derivative term disappears from the volume integral and also after integration by parts the Neumann boundary condition has been introduced on Γ_N.

The equation can be rewritten in bilinear for as:

$$a(w, T) + c(\mathbf{a} \rho c_p; w, T) = (w, s) + (w, h)_{d\Gamma_N} \tag{2.59}$$

where the integrals written in a compact integral form are

$$a(w, T) = \int_\Omega \nabla w \cdot (k \nabla T) d\Omega, \quad (w, s) = \int_\Omega w s d\Omega$$

$$c(\mathbf{a}; w, T) = \int_\Omega w (\mathbf{a} \cdot \nabla T) d\Omega, \quad (w, h)_{d\Gamma_N} = \int_{\Gamma_N} w h d\Gamma$$

The numerical solution of the problem sought for an approximate value of the unknown such that:

$$T(x) \cong \hat{T}(x) \tag{2.60}$$

The approximate solution can be expressed by a linear combination of functions as:

$$\hat{T}(x) = \sum_{i=0}^{n} N_i(x) T_i \tag{2.61}$$

where T_i are the unknown parameters and $N_i(x)$ are the functions of the independent variable x. Expressing the governing equation as $A(T)$ and the boundary conditions of the problem as $B(T)$, we can rewrite the strong form, after use the approximation of the solution, as:

$$\int_{\Omega} wA(\hat{T})d\Omega + \int_{\Gamma} +\bar{w}B(\hat{T})d\Gamma = 0 \tag{2.62}$$

The previous equation is called *weight residual* expression due to $A(\hat{T})$ and $B(\hat{u})$ are the *residual* of the approximation solution in the domain and at the boundaries.

$$\begin{array}{ll} A(\hat{T}) = r_{\Omega} & in \quad \Omega \\ B(\hat{T}) = r_{\Gamma} & in \quad \Gamma \end{array}$$

The value of the residuals indicate the error in satisfaction of the governing differential equations depending of the approximate function $\hat{u}$ [17]. The name weight residual method derives from Eq. 2.62 where the integral of the residual is weighted by the function w.

Choosing a *finite set* of the weighted functions the approximate solution of the differential equation can be obtained and the following system of equations is obtained:

$$\int_{\Omega} wA(\sum_{j} N_j T_j)d\Omega + \int_{\Gamma} \bar{w}(\sum_{j} N_j T_j)d\Gamma = 0; \quad i = 1, n \tag{2.63}$$

The previous system of equation can be rewritten as:

$$\mathbf{KT} = \mathbf{f} \tag{2.64}$$

where $\mathbf{K}$ is a square matrix, $\mathbf{T}$ is the n unknown approximate solution vector and $\mathbf{f}$ is the vector containing the the external loads.

The popular method used is *Galerkin method* which formulate the weight residual problem tacking the approximation function N_i as the weighting function:

$$W_i(x) \equiv N_i(x) \tag{2.65}$$

Using Galerkin method we can rewrite Eq.2.62 as:

$$\int_\Omega N_i A(\hat{T}) d\Omega + \int_\Gamma N_i B(\hat{T}) d\Gamma = 0 \tag{2.66}$$

Spatial Discretization

The concept of spatial discretization is to split the domain under study into a mesh composed by finite elements. For example, in order to obtain the spatial discretization of a two dimensional problem, assuming a spacial discretization of the domain Ω using 2D triangular elements (Figure 2.12) and approximating T inside each element using 2D shape functions $N_i^{(e)}(x, y)$ and defining $T^{(e)}$ as the nodal values of the approximation solutions

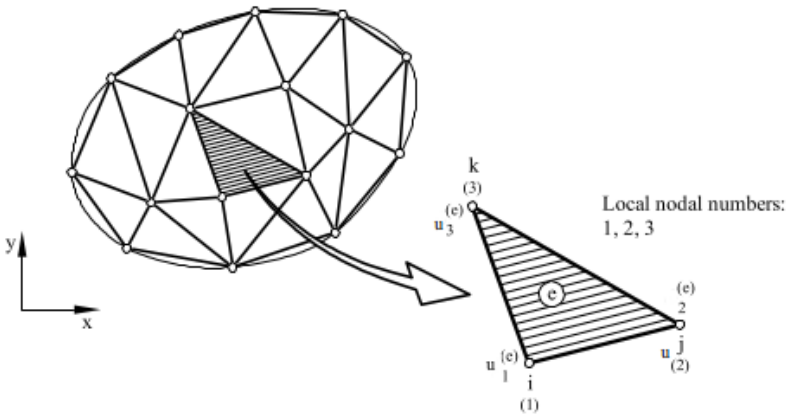


Figure 2.12: 2D domain discretization using three-node triangular elements (Oñate).

we obtain the next system of equations:

$$\sum_e \int\int_{\Omega^{(e)}} \left[\rho c_p N_i^{(e)} \left(a_x \left(\sum_{j=1}^{n} \frac{\partial N_j^{(e)}}{\partial x} T_j^{(e)} \right) + a_y \left(\sum_{j=1}^{n} \frac{\partial N_j^{(e)}}{\partial y} T_j^{(e)} \right) \right) \right] d\Omega$$
$$+ \sum_e \int\int_{\Omega^{(e)}} \left[\frac{\partial N_i^{(e)}}{\partial x} k_x \left(\sum_{j=1}^{n} \frac{\partial N_j^{(e)}}{\partial x} T_j^{(e)} \right) + \frac{\partial N_i^{(e)}}{\partial y} k_y \left(\sum_{j=1}^{n} \frac{\partial N_j^{(e)}}{\partial y} T_j^{(e)} \right) \right] d\Omega \quad (2.67)$$
$$= \sum_e \int\int_{\Omega^{(e)}} N_i^{(e)} q d\Omega + \int_\Gamma N_i^{(e)} \left(k_x \frac{\partial T}{\partial x} n_x + k_y \frac{\partial T}{\partial y} n_y \right) d\Gamma$$

where the sum applied is over all the elements of the domain. Considering the triangular elements used for the discretization and taking into account that due to the type of problem we have one DoF per node, the unknown solution is expressed by the following polynomial:

$$T = \sum_{i=3}^{3} N_i^{(e)} T_i^{(e)} = N_1^{(e)} T_1^{(e)} + N_2^{(e)} T_2^{(e)} + N_3^{(e)} T_3^{(e)} \quad (2.68)$$

The shape functions are identified as:

$$N_i^{(e)}(x, y) = \frac{1}{2A^{(e)}} (a_i^{(e)} + b_i^{(e)} x + c_i^{(e)} y) \quad i = 1, 2, 3 \quad (2.69)$$

where $A^{(e)}$ is the element area and $a_i^{(e)}$, $b_i^{(e)}$ and $c_i^{(e)}$ values are obtained operating with the element nodes coordinates (see [17]). Figure 2.13 shows that linear shape functions take the unit value at the node and zero at the others nodes.

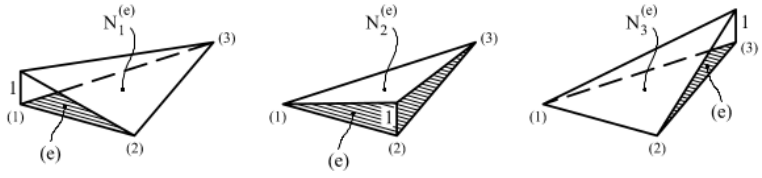


Figure 2.13: Linear shape function for the triangular element (Oñate).

The problem in algebraic form can be rewritten as:

$$[\mathbf{C} + \mathbf{K} - \mathbf{B}_{out}]T = \mathbf{q} \quad (2.70)$$

where $\mathbf{C}$ is the convective matrix, $\mathbf{K}$ is the diffusive matrix, $\mathbf{B}_{out}$ is the external flux matrix, $\mathbf{q}$ is the load vector and they are defined as:

$$C_{ij}^{(e)} = \int\int_{\Omega^{(e)}} \rho c_p N_i^{(e)} \left(a_x \frac{\partial N_j^{(e)}}{\partial x} + a_y \frac{\partial N_j^{(e)}}{\partial y} \right) d\Omega \tag{2.71a}$$

$$K_{ij}^{(e)} = \int\int_{\Omega^{(e)}} \left[\frac{\partial N_i^{(e)}}{\partial x} k_x \frac{\partial N_j^{(e)}}{\partial x} + \frac{\partial N_i^{(e)}}{\partial y} k_y \frac{\partial N_j^{(e)}}{\partial y} \right] d\Omega \tag{2.71b}$$

$$B_{(out)}{}_{ij}^{(e)} = \int\int_{\Gamma^{(e)}} N_i^{(e)} \left(k_x \frac{\partial T}{\partial x} n_x + k_y \frac{\partial T}{\partial y} n_y \right) d\Gamma \tag{2.71c}$$

$$q_i^{(e)} = \int\int_{\Omega^{(e)}} N_i^{(e)} q d\Omega \tag{2.71d}$$

For the derivation of other 2D elements such *quadratic triangular, linear and quadratic quadrilateral* and 3D elements see Oñate.

2.6.3 Weak and Algebraic Formulation of the Navier-Stokes Equations

Applying the same concept in order to obtain the algebraic form of the Navier-Stokes Equations first of all the weak formulation of the problem must be obtained.

Multiplying the linear momentum equation by the test function $\mathbf{w}$, where it can be described as:

$$\mathbf{w} \in \mathscr{V}_0, \quad \mathscr{V}_0 = \{\mathbf{w} \mid \mathbf{w} \in \mathscr{H}_0^1(\Omega)\} \tag{2.72}$$

we obtain:

$$\begin{aligned} \int_\Omega \mathbf{w} \frac{\partial \mathbf{v}}{\partial t} d\Omega + \int_\Omega (\mathbf{v} \cdot \nabla)\mathbf{v} \cdot \mathbf{w} d\Omega - \int_\Omega \mathbf{w} \nu \nabla^2 \mathbf{v} d\Omega \\ + \int_\Omega \mathbf{w} \nabla p d\Omega = \int_\Omega \mathbf{w} \cdot \mathbf{b} d\Omega \quad \forall \mathbf{w} \in \mathscr{V}_0 \end{aligned} \tag{2.73}$$

After integrating by parts the diffusive term and the pressure term as:

$$-\int_\Omega \mathbf{w}\nu\nabla^2\mathbf{v}d\Omega = \int_\Omega \nabla\mathbf{w}\nu\nabla\mathbf{v}d\Omega - \int_\Gamma \nu\mathbf{w}\cdot(\mathbf{n}\nabla\mathbf{v})d\Gamma \tag{2.74a}$$

$$\int_\Omega \mathbf{w}\nabla p d\Omega = -\int_\Omega p\nabla\cdot\mathbf{w}d\Omega + \int_\Gamma \mathbf{w}\cdot(p\mathbf{n})d\Gamma \tag{2.74b}$$

the weak form of the Navier-Stokes Equations is:

$$\begin{aligned}\int_\Omega \mathbf{w}\frac{\partial\mathbf{v}}{\partial t}d\Omega + \int_\Omega(\mathbf{v}\cdot\nabla)\mathbf{v}\cdot\mathbf{w}d\Omega + \int_\Omega\nabla\mathbf{w}\nu\nabla\mathbf{v}d\Omega - \int_\Omega p\nabla\cdot\mathbf{w}d\Omega \\ = \int_\Omega \mathbf{w}\cdot\mathbf{b}d\Omega + \int_\Gamma \nu\mathbf{w}\cdot(\mathbf{n}\nabla\mathbf{v})d\Gamma - \int_\Gamma \mathbf{w}\cdot(p\mathbf{n})d\Gamma \quad \forall\mathbf{w}\in\mathscr{V}_0\end{aligned} \tag{2.75}$$

In the same manner the weak form of the continuity equation of an incompressible flow (Eq. 2.2) can be obtained first of all defining a test function q:

$$q\in\mathscr{Q}_0, \quad \mathscr{Q}_0 = \{q \mid q\in L^2(\Omega)\} \tag{2.76}$$

where the $L^2(\Omega)$ space contains the square integrable functions.

The weak form of the continuity equation can be expressed as:

$$\int_\Omega q\nabla\cdot\mathbf{v}d\Omega = 0 \quad \forall q\in\mathscr{Q} \tag{2.77}$$

The space element discretization of the Navier-Stokes Equations taking the finite element solution as an approximation of the exact solution of a property ($\gamma\equiv\tilde{\gamma}$), Galerkin method states that the velocity interpolation reads as:

$$\tilde{\mathbf{v}}(\mathbf{x},t) = \sum_{j=0}^{n} N_j(\mathbf{x})\mathbf{v}_j(t) \tag{2.78}$$

where $N_j(\mathbf{x})$ are the shape function in Eulerian coordinates. The test function is discretized as:

$$\tilde{\mathbf{w}}(\mathbf{x},t) = \sum_{j=0}^{n} N_j(\mathbf{x})\mathbf{w}_j \tag{2.79}$$

and finally the acceleration is approximated as :

$$\frac{\partial \tilde{\mathbf{v}}}{\partial t}(\mathbf{x},t) = \sum_{j=0}^{n} N_j(\mathbf{x})\dot{\mathbf{v}}_j(t) \tag{2.80}$$

On the other end the pressure is approximated as:

$$p(\mathbf{x},t) = \sum_{j=0}^{n} N_j(\mathbf{x})p_j(t) \tag{2.81}$$

and its test function approximation reads:

$$q(\mathbf{x},t) = \sum_{j=0}^{n} N_j(\mathbf{x})q_j \tag{2.82}$$

After the space discretization is obtained, the momentum equation can be written in algebraic form as:

$$\mathbf{M}\dot{\mathbf{v}} + (\mathbf{K}+\mathbf{C})\mathbf{v} - \mathbf{G}\mathbf{p} = \mathbf{f}^{\mathbf{ext}} \tag{2.83}$$

where $\mathbf{M}$ is the *mass matrix*, $\mathbf{K}$ is the *viscous matrix*, $\mathbf{C}$ is the *convective matrix*, $\mathbf{G}$ is the *pressure matrix* and $\mathbf{f}^{ext}$ is the *volumetric external force vector*.

For a 2D case the coefficient are computed as:

$$M_{ij}^{(e)} = \int\int_{\Omega^{(e)}} N_i^{(e)} N_j^{(e)} d\Omega \tag{2.84a}$$

$$K_{ij}^{(e)} = \nu \int\int_{\Omega^{(e)}} \left(\frac{\partial N_i^{(e)}}{\partial x}\frac{\partial N_j^{(e)}}{\partial x} + \frac{\partial N_i^{(e)}}{\partial y}\frac{\partial N_j^{(e)}}{\partial y} \right) d\Omega \tag{2.84b}$$

$$C_{ij}^{(e)} = \int\int_{\Omega^{(e)}} N_i^{(e)} \left(u\frac{\partial N_j^{(e)}}{\partial x} + v\frac{\partial N_j^{(e)}}{\partial y} \right) d\Omega \tag{2.84c}$$

$$G_{ij}^{(e)} = \int\int_{\Omega^{(e)}} \left(\frac{\partial N_i^{(e)}}{\partial x} + \frac{\partial N_i^{(e)}}{\partial y} \right) N_j^{(e)} d\Omega \tag{2.84d}$$

$$f_i^{ext}\ (e) = \int\int_{\Omega^{(e)}} N_i^{(e)} (b_x + b_y) d\Omega \tag{2.84e}$$

The algebraic form of the continuity equation reads:

$$\mathbf{G}^T \mathbf{v} = 0 \tag{2.85}$$

Where the $\mathbf{G}$ is the *divergence matrix* which is equal to the transpose of the pressure matrix, where its coefficients are obtained as:

$$G_{ij}^{T\ (e)} = \int\int_{\Omega^{(e)}} N_i^{(e)} \left(\frac{\partial N_j^{(e)}}{\partial x} + \frac{\partial N_j^{(e)}}{\partial y} \right) d\Omega \tag{2.86}$$

2.6.4 LBB condition

The LBB condition, as well known as *inf-sup* condition, states that a link between pressure and velocity spaces must be defined. A necessary but not sufficient condition in order to determinate an unique solution for $\mathbf{v}$ and p which states a necessary relation between finite element pressure and velocity dimension spaces is

$$dim\mathscr{Q}^h \leq dim\mathscr{V}^h$$

So the existence of a stable approximation of the numerical solution $(\mathbf{v}^h, p^h)$ depends on the discretized spaces $\mathscr{V}^h$ and $\mathscr{Q}^h$, such as follow:

$$inf_{q^h \in \mathscr{Q}^h} \quad sup_{\mathbf{w^h} \in \mathscr{V}^h} \frac{q^h, \nabla \cdot \mathbf{w^h}}{\|q\|_0 \|\mathbf{w^h}\|_0} \geq \alpha > 0 \tag{2.87}$$

where α is independent of the mesh size. The uniqueness existence of $v^h \in \mathscr{V}^h$ and $p^h \in \mathscr{Q}^h$ is ensured if the LBB condition is satisfied.

2.7 Time Discretization

2.7.1 Explicit vs. Implicit Schemes

Explicit schemes are easy to code/parallelize but the time step may not exceed a certain threshold that depends on the Peclet number and on the smallest mesh size. The inconvenient of the explicit schemes are lack of stability and, for extremely small time steps, a high computational cost. Implicit schemes implies a large cost per time step because a system of equations at every time step has to be solved. On the other hand, most implicit schemes are unconditionally stable, and the use of large time steps makes it possible to reach the final time faster than with an explicit scheme subject to a restrictive stability limit. Due to the unsteady behaviour of the problem which involves a time dependency term, a time discretization method has been used. Several methods are available and next the θ *scheme time integration*, *Backward differentiation in time* and *Adams method (Linear multistep method)* will be discussed. In the current work the BDF2 second order method has been used in order to approximate the time derivative operator.

2.7.2 θ Scheme Time Integration

The θ methods are based on the formula

$$\frac{T^{n+1}-T^n}{\Delta t} = \theta T_t^{n+1} + (1-\theta)T_t^n + \mathcal{O}((1/2-\theta)\Delta t, \Delta t^2)$$

The formula can be rewritten after truncation error as

$$\frac{\Delta T}{\Delta t} - \theta \Delta T_t = T_t^n$$

where

$$\Delta T = T^{n+1} - T^n \quad and \quad \Delta T_t = T_t^{n+1} - T_t^n$$

For the heat equation the time discrete scheme is:

$$\begin{aligned} \rho c_p w \frac{\Delta T}{\Delta t} + \theta[\rho c_p w \mathbf{a}\cdot\nabla - \nabla\cdot(k\nabla)]\Delta T = \theta q^{n+1} \\ + (1-\theta)q^n - [\rho c_p w \mathbf{a}\cdot\nabla - \nabla\cdot(k\nabla)]T^n \end{aligned} \tag{2.88}$$

Taking $\theta = 0$ the *Forward Euler* method, which is the explicit method of the family, is recovered. In order to use an implicit scheme, the *Crank-Nicolson* method is obtained taking $\theta = 1/2$ and the *Backward Euler* method taking $\theta = 1$. Among the implicit methods, Crank Nicolson is the only second order accurate.

The matrix form of the problem is:

$$\mathbf{M}\delta_t(T^{n+1} - T^n) + \theta(\mathbf{C} - \mathbf{K})\Delta T = \theta q^{n+1} + (1-\theta)q^n - (\mathbf{C} - \mathbf{K})T^n \tag{2.89}$$

Where $\mathbf{M}$ is the mass matrix, $\mathbf{C}$ is the convective matrix and $\mathbf{K}$ is the diffusive matrix.

2.7.3 Backward Differentiation in Time

The method adopted is called *Backward differentiation in time* and several previous steps in time are used to approximate the time derivative at t^{n+1}. Using this method different schemes can be used and all of them are implicit because the operator at t^{n+1} is evaluated and the method results unconditionally stable for any size of the time step. Considering the heat equation without stabilization, the problem in algebraic form is:

$$\mathbf{M}\delta_t(T^{n+1} - T^n) + (\mathbf{C} - \mathbf{K})T^{n+1} = q \tag{2.90}$$

The approximation for $\partial_t T^{n+1}$ can take different order:

- $BDF1 : \delta_t T^{n+1} = (T^{n+1} - T^n)/\delta_t$
- $BDF2 : \delta_t T^{n+1} = (\frac{3}{2}T^{n+1} - 2T^n + \frac{1}{2}T^{n-1})/\delta_t$
- $BDF3 : \delta_t T^{n+1} = (\frac{11}{6}T^{n+1} - 3T^n + \frac{3}{2}T^{n-1} - \frac{1}{3}T^{n-2})/\delta_t$

BDF1 method is unconditionally stable of order 1 ($\mathcal{O}(\delta t)$) and BDF2 method is also unconditionally stable but with higher order up to 2 ($\mathcal{O}(\delta t^2)$). BDF3 method has an higher order, up to 3 ($\mathcal{O}(\delta t^3)$), but is conditionally stable.

2.7.4 Adams Method

The Adam method is a second order accurate derived from polynomial fitting and it can be either explicit or implicit. For the explicit scheme the Adam method follow:

- $ADB1 : \mathbf{M}/\delta_t(T^{n+1} - T^n) + (\mathbf{C} - \mathbf{K})T^n = q^n$
- $ADB1 : \mathbf{M}/\delta_t(T^{n+1} - T^n) + \frac{3}{2}(\mathbf{C} - \mathbf{K})T^n - \frac{1}{2}(\mathbf{C} - \mathbf{K})T^{n-1} = \frac{3}{2}q^n - \frac{1}{2}q^{n-1}$

And for the implicit scheme the method follow:

- $ADB1 : \mathbf{M}/\delta_t(T^{n+1} - T^n) + (\mathbf{C} - \mathbf{K})T^{n+1} = q^{n+1}$
- $ADB1 : \mathbf{M}/\delta_t(T^{n+1} - T^n) + \frac{1}{2}(\mathbf{C} - \mathbf{K})T^{n+1} + \frac{1}{2}(\mathbf{C} - \mathbf{K})T^n = \frac{1}{2}q^{n+1} - \frac{1}{2}q^n$

Chapter 3

Methodology

In this chapter are collected all the main aspects which compose the current work.

The first section introduces Kratos software and also all the steps in order to implement the application needed in order to solve a thermo-fluid dynamic problem. Than the strategy used concerning the type of the problem where different fields are coupled together is described and also the Dirichlet-Neumann method, which has been used in order to iterate between subdomains, is introduced. Next the 3D heat exchanger problem and modelling are introduced and finally a brief introduction and description related with heat exchanger is presented.

3.1 Kratos Muliphysics Software

Kratos is a framework for building multi-disciplinary finite element program composed by a wide variety of applications involved in different engineering areas.

The flexibility of the framework is improved adopting Python scripting language in order to define the main procedure of Kratos.

Kratos can be used from a developer or user point of view. Finite element developers can take the advantages of all the Kratos's requirements even without having a deep C++ programming knowledge and also, at the same time, Kratos may be used by application developers in order to optimize programs or design tools without having a strong interest in finite element programming. Users of Kratos are more interested in model and solve engineering problems without getting involved in programming. In this case Kratos offers a flexible external interface without implementing any programming code.

The main features of Kratos are:

- MULTI-PHYSIC: Multi-Physic problems are one of the most important topic nowadays. A multi-physics simulation involves multiple physical models or multiple simultaneous physical phenomena.

- FINITE ELEMENT METHOD (FEM): FEM is a powerful method extremely popular in order to solve Partial Differential Equations (PDE) using the numerical analysis.
- OBJECT ORIENTED PROGRAMMING (OOP): OOP is a programming paradigm based on the concept of objects, which may contains attributes and procedures. Due to the flexibility, the modular design, abstraction and hierarchy, this methodology is needed in order to maximum exploit the numerical methods techniques.
- OPEN SOURCE: An open source software is developed and share for free. It is focused in the main code free access.
- FREE: It is free with the purpose to share knowledge and built a robust numerical methods and also everyone is able to modify and distribute the software.

3.1.1 Problem Description and Main Kratos's Tools

As explained before, Kratos uses finite element model formulation in order to solve multi-physics problems where routines such as assembling matrices, solving systems, printing results, and others tasks are involved in the framework.

An application has been generated using all the tools provided by Kratos after implementing an **element**, a **condition** and a **solver** in order to solve a physical problem.

The application created solves a transient heat equation problem for a 2D/3D case (one degree of freedom (DoF) per node) and the mathematical formulation respects equation 2.14.

The problem consists of different components, as the stiffness matrix **K**, the **Dirichlet boundary conditions**, the **Neumann boundary conditions** and the **thermal loads**.

Due to the rigidity matrix will be calculated element by element, an element must be implied in the application. The Dirichlet conditions has to be multiplied by the K matrix and it will be done element-wise. Finally, the thermal loads will add them node by node to the right hand side vector of the system of equation **(RHS)**. Also a condition, in order to take into account the contributions of the **Neumann boundary conditions**, has been created.

In order to execute Kratos and execute a problem type, python's scripts are used and the following tasks are done:

1. Load Kratos and import applications
2. Read the problem data and create the *model part* in Kratos

3. Define the *builder* and *solver* used
4. Call the solver
5. Write a file with the mesh and results

The files needed in the application are:

- **HeatEquationSolver.py**
- **HeatEquationApplication.h** and **HeatEquationApplication.cpp** to define the application
- **HeatEquation.h** and **HeatEquation.cpp** to define the element
- **Extra files** such as **.py** and **cmake**

3.1.2 Creating an Element

Elements are created using a so called *symbolic generation* implementation. The idea is to generate a Kratos element which includes triangular and tetrahedron linear elements.

The **HeatEquation.cpp** script, which describes the element used in the application, has been generated using the next scripts:

- **GenerateHeatEquationElement.py**
- **HeatEquationTemplate.cpp**
- **HeatEquation.h**

GenerateHeatEquationElement.py

The *GenerateHeatEquationElement.py* script permits to create the element and it contains all the *symbols definitions* (see A).

The symbol definitions includes the material poperties, i.e. the thermal conductivity, the *unknown fields*, (in this application the unknown filed is the temperature and it has been defined for different time steps), the *shape functions*, the *shape functions derivatives*, and the *field test functions* (temperature field test function).

Once the previous symbolic definitions has been implemented, the data interpolation to the Gauss point are computed and also the *gradients fields* are enforced.

Once all the terms of the weak form evaluated at the Gauss points are defined, finally the *functional*, the *RHS* and the *LHS* are implemented. The *ComputeRHS* function computes the differentiation of the *residual* depending on the *test function* and *ComputeLHS* function computes the differentiation of the *RHS* depending on the *DoF*, which in this case is the temperature.

HeatEquationTemplate.cpp

The *HeatEquationTemplate.cpp* incorporates the *EquationIdVector* and the *GetDofList* functions (see B). The first function returns a vector with the list of "EquationIds" associated to each element and those ids are intended as the position of the corresponding *DoF* in the global system matrix. The second function creates the vector which contains the unknown field results for each element.

The script contains all the functions for both three and two dimensions.

HeatEquation.h

The *HeatEquation.h* class stores the *CalculateLocalSystem* and the *CalculateRightHandSide* methods, where the first one compute the contribution to the global system and the second one's task is to obtain the *right hand side* vector (both for each element).

The *HeatEquation.h* class also includes the *FillElementData* method which assigns all the properties, i.e. the density, and the unknown solution to each element (see G).

Finally, through *GetShapeFunctionsOnGauss* methods, the shape functions values at each Gauss points are given for both 2D and 3D cases,

HeatEquation.cpp

The current script follows the *HeatEquationTemplate.cpp* structure with including all the operations involved in the symbolic implementation of the functional and in the residual, both defined in the *GenerateHeatEquationElement.py*. The implementation concern the *ComputeGaussPointLHSContribution* and the *ComputeGaussPointRHSContribution* functions (see F). Both functions compute the Gauss points at the left and right hand side.

3.1.3 Application Solvers

Heat Equation Solver

The solver script's purpose is to act as the interface between the python script file [Sect.3.1.5] and the Kratos functions. The solver is a class composed by different methods already customized able to solve the problem where the main parts are introduced next (see C).

An important method already implemented in Kratos is the type of solver to use and, between all types for this specific problem, a *SuperLU* strategy is adopted

It is also here where all the variable, i.e. the density, and the DoF, where in this specific problem is the temperature, are incorporated.

In the solver are also included the functions *ImportModelPart*, where the *model part* is raed, and the function *ExecuteAfterReading*, where the elements and conditions are replaced.

Than it is here where the *initialize* function is created. In this function the solution strategy is created defining the tolerances convergence criteria, computing the time scheme coefficients and defining the numerical analysis method, which in this case is Newton-Raphson.

Also the function *ModelPartReading*, which reads the model part in the *mdpa* file, and the function *SetBufferSize*, which set the dimension of the buffer size, are implemented in the solver. Than the function *ExecuteAfterReading* permits Kratos to gets and set the model parameters defined in the *mdpa*.

CFD Heat Transfer Solver

Considering the nature of the problem the idea has been to create a solver which inserts both the fluid dynamic solver and the heat equation solver (see D).

The *CFD heat transfer solver* follows more nearly the structure of the previous solver but with the addition of several implementations.

First of all both solvers has to be imported and created using the *import* function.

After add all the variables, DoF and import the model parts, the next to methods have been used: the *ConnectivityPreserveModeler* and *GenerateModelPart*.

The ConnectivityPreserveModeler function takes every fluid conditions and creates another condition for the thermal part of the model. The thermal element can see every fluid properties because they share the same nodes and, as the fluid velocity change, the thermal part can see it. Finally the fluid part has to be solved and than the thermal part of the model.

3.1.4 Application Conditions

Heat Equation Neumann Condition

Once the solver is called it will ask all the nodes contributions and nodes conditions (see E). In order to compute all the contributions to the global system, it has been programmed in the *CalculateLocalSystem*. Here all the necessary computations are performed and the LHS and RHS are returned.

Besides returning the 1x1 matrix and the RHS vector, we must tell the builder which degrees of freedom each of the lines of this matrix must add its contribution using both *EquationIdVector* and *GetDofList*.

FACE HEAT FLUX is the variable which defines the external fluxes and in order to include them, the *Heat Equation Neumann Condition* has been created. KRATOS will call the conditions and assigns them node by node to the system of equations.

3.1.5 Creating a Problem to Be Solved

Once the application is created, in order to test it, a very simple problem has been created. When an example from the interface of Kratos is lunched, this is what is going on behind: first it creates a geometry file that Kratos can understand, the *.mdpa* file , and then it executes a python script that contains all the instruction for kratos. The two files created are:

- *modelpart* (*.mdpa*) file defining the properties, geometry, BCs,...
- python file able to compile the *mdpa* through Kratos

Modelpart File

An example of a *mdpa* file corresponding to a square domain composed by 18 triangular elements (Figure 3.1) is desrcibed next.

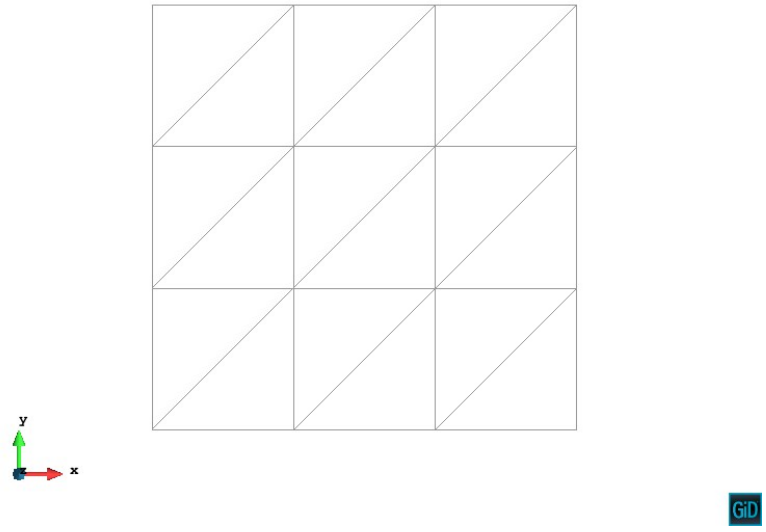


Figure 3.1: Square domain 18 triangular elements.

The values of the properties are introduced as follow and also a *properties group* must be created, followed by a number which defines the group

```
Begin Properties 1
    CONDUCTIVITY 1.0
    SPECIFIC_HEAT 1.0
    DENSITY 1.0
End Properties
```

The nodes are introduced identifying first the node number and than the x, y, z coordinates

```
Begin Nodes
    1   0.0000000000   1.0000000000   0.0000000000
    2   0.0000000000   0.6666666667   0.0000000000
    3   0.3333333333   1.0000000000   0.0000000000
    ...
End Nodes
```

The elements are defined by the element identification (first column), then the element properties, and finally the nodes which compose the elements. The name of the element is *HeatEquation2D* and it is implemented as

```
Begin Elements HeatEquation2D
     1          1         12          7         10
     2          1          7          5         10
     ...
End Elements
```

As the elements, conditions are defined first with the identification, than the properties and finally the nodes to apply the condition. The condition *Condition2D2N* refers a two nodes line and it is implemented as

```
Begin Conditions Condition2D2N
    1 0 10 12
    2 0 12 14
    ...
End Conditions
```

Finally the boundaries are defined creating a *SubModelPart* where first identifying the nodes and than the number of the conditions.The *Inlet2DBoundary* is defined as

```
Begin SubModelPart Inlet2DBoundary
Begin SubModelPartNodes
     1
     2
    ...
End SubModelPartNodes
Begin SubModelPartConditions
     1
     2
    ...
End SubModelPartConditions
End SubModelPart
```

3.2 Strategy

3.2.1 Transmission Conditions

The thermo-fluid dynamic problem, where conduction and convection are involved, has the difficulty to analyze the occurrence at the interface, where certain transmission conditions have to be accomplished.

Let consider a problem domain divided into two disjoint subdomain, such that $\Omega = \Omega_1 \cup \Omega_2$, being Ω_1 the solid subdomain and Ω_2 the fluid subdomain, and both of them are separated in Γ_{int} by the interface. Γ_u and Γ_n are the boundaries where the Dirichlet and the Neumann are defined.

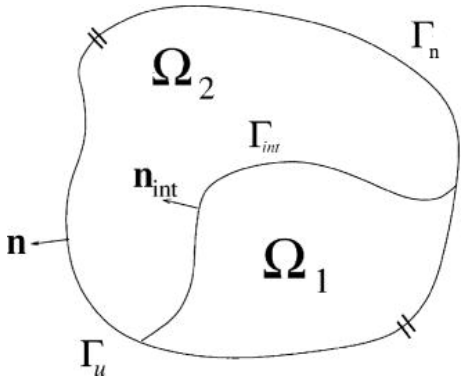


Figure 3.2: Physical domain decomposed into two subdomains (Ω_1, Ω_2) separated by the interface Γ_{int} (Chessa).

Due to the solid fixed position ($\mathbf{v}_s = 0$) and assuming that the mechanical tractions produced by the fluid on the solid are small, it implies that the *mechanical problem is uncoupled and the coupling is due to the thermal problem only.*

In order define a properly communication of the information at the interface, the *transmission conditions* in Γ_{int} are defined and they are:

- continuity of the solution:

$$T_1 = T_2 \quad on \quad \Gamma_{int} \tag{3.1}$$

- continuity of the normal fluxes:

$$k_1 \frac{\partial T}{\partial n_1} = k_2 \frac{\partial T}{\partial n_2} \quad on \quad \Gamma_{int} \tag{3.2}$$

The first transmission condition states that the jump of the solution at the interface must be zero and mathematically can be expressed as:

$$[\![T]\!]_\Gamma = 0 \quad on \quad \Gamma_{int} \tag{3.3}$$

The second transmission conditions defines that the normal component of the fluxes are weakly continuous at the interface and the jump of the flux is equal to zero on: Γ_{int}

$$[\![k\frac{dT}{dn}]\!]_\Gamma = 0 \quad on \quad \Gamma_{int} \tag{3.4}$$

At the interface the first transmission condition is usually enforced in a strong manner, meanwhile the second transmission condition is imposed in a weak sense.

3.2.2 Approach Analysis

There are two many approaches in order to solve the thermo-fluid dynamic problems:

- Monolithic approach: the Navier-Stokes Equations and the heat equations are solved simultaneously using one solver.
- Partitioned approach: the Navier-Stokes Equations and the heat equation are solved separately using different solvers.

Monolithic approaches presents several advantages as the coupled problems are solved in single step and no convergence or iterative issues are needed. Unfortunately the global system of equations to be solved can be very large and also it can be ill-conditioned due to the fact that different physical parameters, constitutive laws and psychical problems are involved.

Modularity and customization are ones of the advantages of partitioned schemes. Each problem and all the implementations can be packed in a software module and each field can be performed with the better suitable algorithm. Also partitioned scheme make easy problems where non-matching grids are involved.

Due to the fact that both the fluid dynamic problem and the thermal problem occur in the same domain (fluid domain), and also because of the non-linearity of the temperature term in the heat equation, it is not possible to solve the global system in a single step.

The best strategy is first solve the velocity and pressure for fluid dynamic problem, and than plug the velocity into the heat equation in order to solve the temperature. Once the temperature is resolved, it is used to compute the Boussinesq body forces on the fluid.

3.2.3 Dirichlet-Neumann Method

The strategy to solve the problem involving two different continuums, i.e. the *Laminar Flow Around a Solid* (Sect.4.3.2), where heat is transferred from a cylinder to a fluid flowing around it, the problem can split it in two subdomains (Ω_s and Ω_f) by assigning the transmission conditions (Eq. 3.1 and Eq. 3.2) at the interface Γ. The iterative algorithm can be set up following the *Dirichlet-Neumann problem* (DN): considering F as the fluid mathematical operator, S as the solid mathematical operator and given $u_2^{(0)}$ on Γ, for k ≥ 1 solve the problems:

$$\begin{cases} Fu_1^{(k)} = f \quad on\ \partial\Omega_f \\ u_1^{(k)} = u_2^{(k-1)} \quad on\ \Gamma \\ u_1^{(k)} = 0 \quad on\ \partial\Omega_f \backslash \Gamma \end{cases} \qquad \begin{cases} Su_2^{(k)} = f \quad on\ \partial\Omega_s \\ \frac{u_2^{(k)}}{\partial n} = \frac{u_1^{(l)}}{\partial n} \quad on\ \Gamma \\ u_2^{(k)} = 0 \quad on\ \partial\Omega_s \backslash \Gamma \end{cases}$$

where depending on l the scheme can be:

- $l = k - 1$ (parallel scheme)
- $l = k$ (sequential scheme)

Condition 3.1 is used as a Dirichlet boundary condition on the interface Γ for the sub-problem Ω_s meanwhile condition 3.2 is used as a Neumann boundary condition at the interface Γ for the sub-problem Ω_f where it has to be prescribed weakly.

3.3 3D Heat Exchanger Modeling

3.3.1 Problem Description and Properties

The problem involves an incompressible fluid entering from the heat exchanger's nozzle (inlet) describing a paraboloid profile depending on the fluid mean velocity. In order to have a fully developed turbulent regime , the mean velocity flow U_m starts from 0 m/s and reaches 5 m/s (Re = 81551) using a sinusoidal velocity function (Eq.4.10g) in the time interval $0 \leq t \leq 1$ s. After 1 seconds the fluid velocity is kept constant (Eq.4.10h).

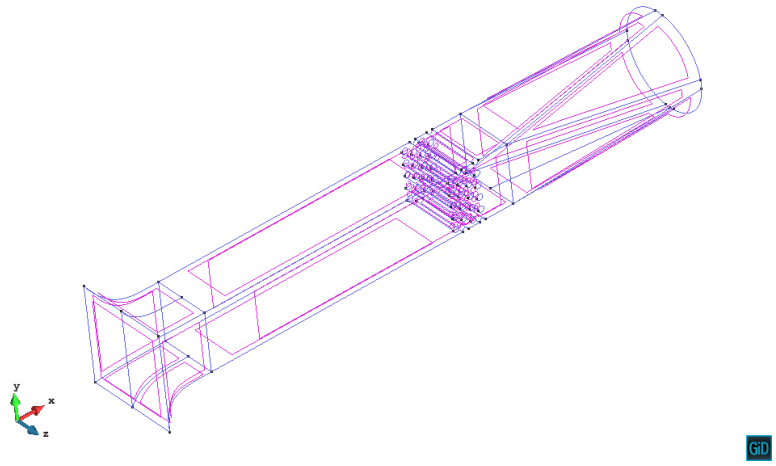
The fluid initial temperature is $T = 298.15$ K and the solid initial temperature is $T = 353.15$ K. Also at the inlet the fluid temperature has been fixed to 298.15 K during all the simulation.

The fluid in question is an incompressible Newtonian fluid where the kinematic viscosity $\nu = 1.47760E^{-5}\ m^2/s$, the fluid density is $\rho = 1.205\ kg/m^3$, the thermal

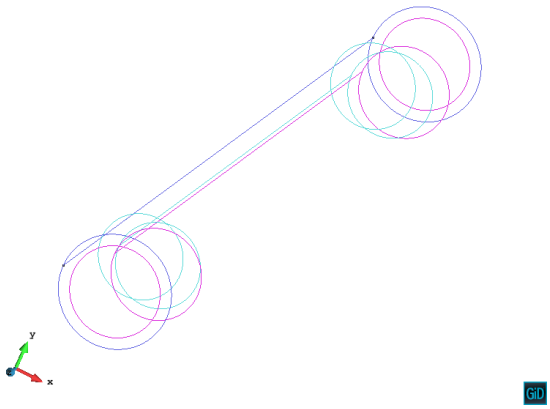
conductivity k= 0.0257 $W/(mK)$ and the specific heat c_p= 1.005 $kJ/(kgK)$. For the solid the density is ρ= 8940 kg/m^3, the thermal conductivity k= 401 $W/(mK)$ and the specific heat c_p= 0.39 $kJ/(kgK)$.

3.3.2 Geometry

The 3D heat exchanger geometry has been created using *Grasshopper* and than, in order to create the model, it has been imported in GiD . The model consists in two different geometries: the body of the heat exchanger and the heated cylinder. Both have been set separately treated as two different domains. It has not been taking into account the presence of all the cylinders but only one of them heated. The heated cylinder is located at the first cylinder's row and the no heated cylinders are considered as an empty space with no material properties.



((a)) Heat exchanger body.



((b)) Cylinder.

Figure 3.3: Geometry domains in GiD.

3.3.3 Boundaries Conditions and Initial Conditions

Due to the problem involves two different physics, the conditions can be divided into two groups: the *fluid mechanics conditions* and the *thermal conditions*.

The first group includes all the conditions at the boundaries related with pressure and velocities: the inlet (inflow condition), the outlet (outflow condition), the walls and the cylinder interface conditions.

At the outlet, in order to do not have any inflow situation due to the vortex generated, all the velocity's components have been set equal to zero but the x component, which has been imposed to be great than zero. Also at the outlet, due to the flow experiments buoyancy effects, the hydrostatic pressure is set as the boundary conditions.

The walls and cylinder interface conditions are characterized by the no-slip condition, where the velocity is imposed equal to zero.

The second group includes the boundary conditions and initial conditions, where the fluid and the cylinder temperature are set at t=0 and the inlet fluid temperature is fixed at the inlet.

Once the models have been imported in GiD, all the boundary conditions have been assigned according to the type of the problem. All the boundary conditions and the mathematically description of the problem are described in Sec. 4.6.

The boundary conditions must be selected for each surface, i.e. the inlet of exchanger's body (Fig.3.4).

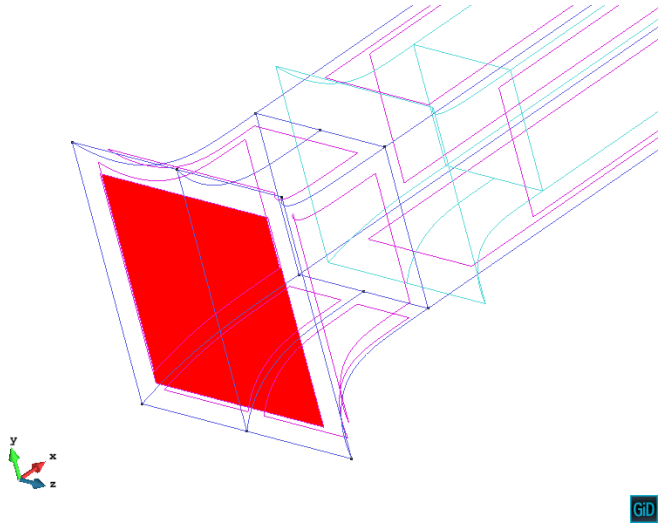


Figure 3.4: Inlet boundary condition definitions in GiD.

3.3.4 Mesh Generation

Once all the boundary condition have been set, the next step has been the mesh generation.

Gid offers the possibility to generate different type of meshes: *structured*, *unstructured* and *semi-structured*. The structured mesh differentiates from the unstructured due to the internal node is connected with a constant number of elements.

In the current model an unstructured mesh has been generated taking into account that near the walls and at the interface between the cylinder and the heat exchanger's body, the mesh has to be very dense in order to have an accurate solution (Fig.3.5)

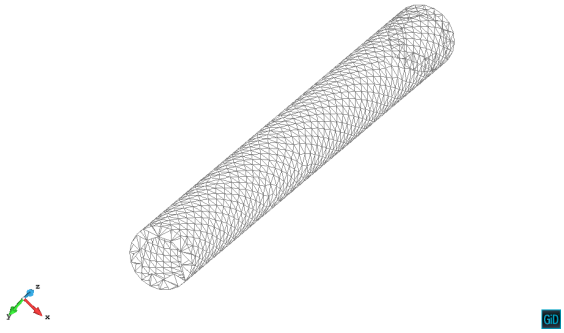


Figure 3.5: Cylinder meshed.

3.4 Solution at the Interface

In order to transfer all the information between the solid and the fluid, a function called $NonConformantOneSideMap$ has been used (see [21]).

Specifically the function couples the fluid and solid nodes at the interface and the problem is solved using the Dirichlet-Neumann scheme.

Once the mapper has been set, the next to functions $StructureToFluidScalarMap$ and $StructureToFluidScalarMap$ pass the variables informations from one filed to the other.

The solution for each time steps has been obtained using a Dirichlet-Neumann routine where first the fluid has been solved and than, according to a certain tolerance, the solid and the fluid have been iterating. During the iteration, once the temperature solution at the solid boundaries has been computed, it has been used has a Dirichlet condition for the fluid at the interface. Next step has been to solve the fluid and computing the thermal fluxes at the interface. Those thermal fluxes has been used as a Neumann condition for the solid and the same routine has been used for each time step (see Algo. 1).

Algorithm 1 Dirichlet-Neumann scheme

1: **for** t$< t_{end}$ **do**
2: *Solve the Fluid* $\rightarrow$ *compute the fluxes* $F_{\Gamma,f}$
3: **while** $T_{\Gamma,s} - T_{\Gamma,f} <$ *tolerance* **do**
4: $F_{\Gamma,s} = F_{\Gamma,f}$ ▷ assign Neumann BC for the solid
5: *Solve the Solid* $\rightarrow$ *compute the temperature* $T_{\Gamma,s}$
6: $T_{\Gamma,f} = T_{\Gamma,s}$ ▷ assign Dirichlet BC for the fluid
7: *Solve the Fluid* $\rightarrow$ *compute the fluxes* $F_{\Gamma,f}$

3.5 Boussinesq Approximation

According to the Boussinesq approximation (Sec.2.1.5), due to the change in temperature of the fluid, at each time step the gravity has been compute following the next algorithm (see Algo. 2):

Algorithm 2 Boussinesq Approximation

1: **for** t< t_{end} **do**
2: *get temperature solution* T_f
3: $g_y = g_0 * (1 - 1/T_{f,0} * (T_f - T_{f,0}))$ ▷ compute the new gravity value g_y
4: $g = g_y$ ▷ set g_y as new gravity value

3.6 Heater Exchangers

Heat exchangers are devices able to exchange energy due to a temperature difference between two different mediums. Convection and diffusion are the process involved in order to transfer the The rate of heat transfer between the two fluids at a location in a heat ex- changer depends on the magnitude of the temperature difference at that location, which varies along the heat exchanger [11].

3.6.1 Types of Heat Exchangers

According to the application, heat exchangers can be classified depending the types of hardware and configuration as a *double-pipe* or *compact*.

The double pipe heat exchanger has the simplest design (Figure 3.6) and it consists of two concentric pipes of different diameter, where one fluid flows in the inner pipe mean while the other fluid flows in the annular space between the two pipes. According to the flow direction, the double pipe heat exchanger can have a parallel flow arrangement, where both fluids flow in the same direction, or in counter flow, which is the case where both fluids have the opposite direction.

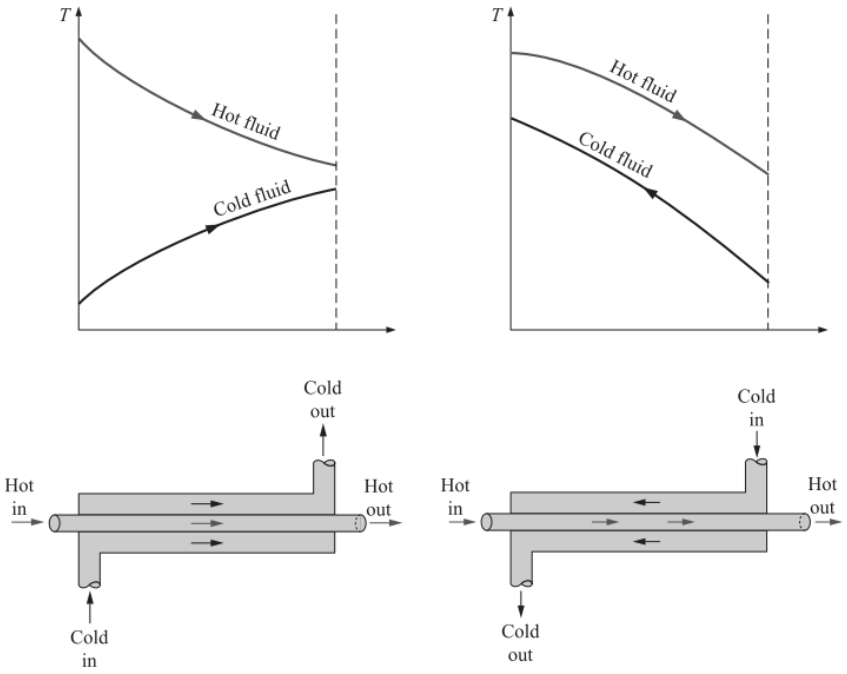


Figure 3.6: Double pipe heat exchanger design. Parallel flow (left) and counter flow (right) (Cengel).

The compact heat exchanger design leads to realize a large heat transfer surface per unit of volume. It is a more complex design compared with the double pipe and it fits for the case where the weight and the volume of the heat exchangers restrict the application possibility. The ability to obtain a large surface area is obtained by attaching closely spaced thin plate to the walls separating the two fluids. When the two fluids flows are perpendicular the compact heat exchanger is called cross-flow. Depending on the flow configuration, the flow configuration can be classified as mixed and unmixed. In the first case (Figure 3.7.a) the fluid is free to move in the transversal direction meanwhile for the unmixed case (Figure 3.7.b) the fluid is prevented to move in the transversal direction.

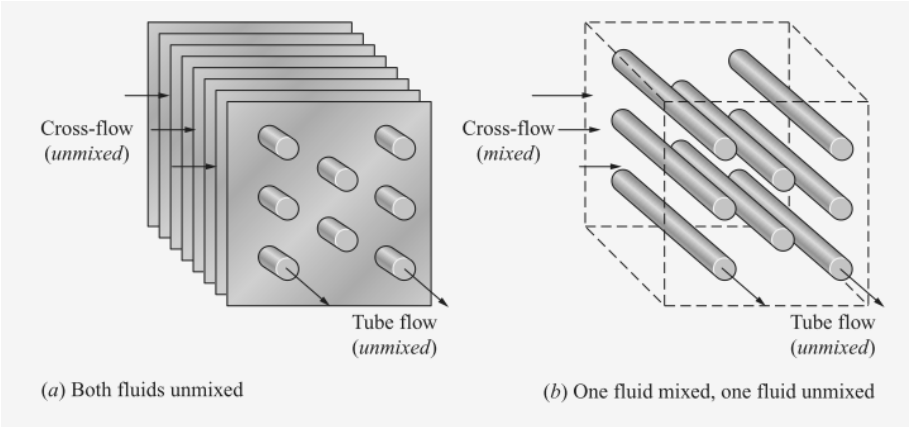


Figure 3.7: Cross-flow unmixed and mixed configuration (Cengel).

3.6.2 Cross-Flow Heat Exchanger

The EUETIB fluid mechanic laboratory heat exchanger (Figure 3.8) is a compact cross flow mixed arrangement type and its utility has been to analyze the convection phenomena between a air at room temperature and a cylinder made of cooper at 80.

The device consists of the next elements:

1. Support bench
2. Nozzle (square-section)
3. Suction tube (square-section)
4. Acrylic test area
5. Diffuser
6. Volute chamber
7. Pipe output
8. Valve regime control
9. Pitot tube
10. Manometer
11. Electrical resistance
12. Thermocouple
13. Thermocouple
14. On/Off

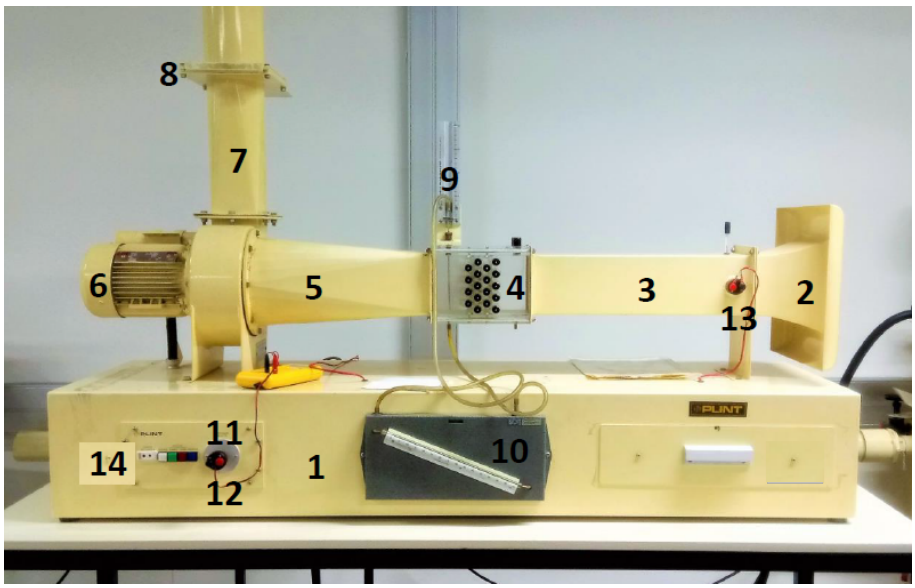


Figure 3.8: Cross-flow heat exchanger EUETIB FM Laboratory (Riera M, 2016).

Heat Exchanger Under Operation

When the device is operating, the air is moved by suction because of the work done by the volute chamber. The fluid enters from the nozzle, which is a device designed in order to increase the velocity and to have a better control of the direction of the fluid. As the nozzle section decreases, the fluid experiment a kinetic energy increasing and therefor a pressure decreasing. As the fluid circulates through all the suction tube, the air profile velocity develops until it arrives at the test area (Figure 3.9), where it finds a group of tubes perpendicular to the flow direction.

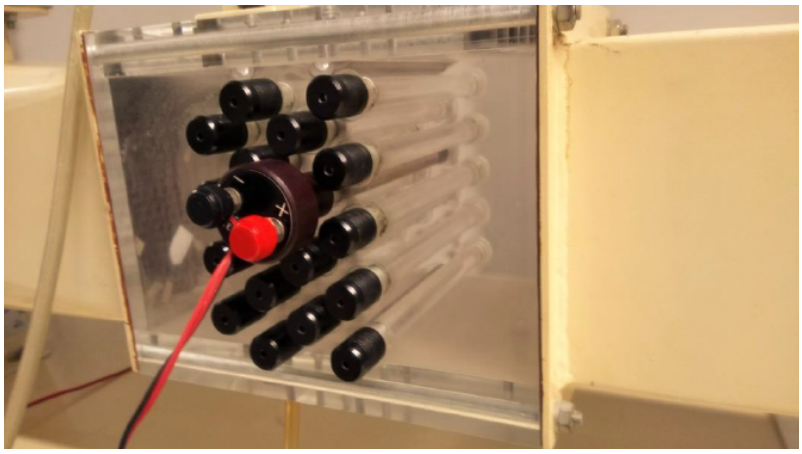


Figure 3.9: Methacrylate cylinders heat exchanger test area (Riera M, 2016).

In order to measure the pressure at different areas, a Pitot tube is installed. Its utility is important especially to collect data before and after the test area, where the fluid interacts with the cylinders in order to study the fluid velocity profile. Under the test area it is also present a manometer used to measure the dynamic pressure.

Once the fluid have passed over the test area it circulates through a diffuser, which is function is to decrease the velocity before to arrive at the volute chamber.

In order to study different flow regimes, a valve (Figure 3.10), which controls the amount of air flow inside the heat exchanger, is positioned on the top of the diffuser at the output pipe.

Figure 3.10: Fluid flow control valve (Riera M, 2016).

The heat transfer is simulated by substituting one of the methacrylate cylinder by one made of copper, which it has been warmed up through an electrical resistance reaching a temperature of $80°C$. The methacrylate material is optimal considering its thermal properties. In contradistinction to copper, which is a thermal conductor, methacrylate is designed in order to do not condition the heat flow by limiting heat conduction and convection.

The device also disposes of two thermocouples, one at the nozzle and another one in the interior of the copper cylinder. The thermocouples return the value of the temperature difference between the air at the inlet and the cylinder copper. The result is returned in mV due to a multimiter. The conversion from mV to grade $°C$ is $1°C = 0.041\ mV$

Chapter 4

Tests and Results

In this chapter all the simulations carried out are collected in order to test the reliability of the Heat Equation Application. In section 4.1 the heat equation is tested for next 2D cases: the *pure diffusion*, the *diffusion dominant* and the *convection dominant*.

The test of the solver *CFD-Heat Transfer*, which couples the heat equation and the fluid dynamic application in 2D, is presented in section 4.3 and than the numerical simulation of the heat exchanger in 2D is provided in section 4.4. Finally in section 4.5 the 3D test of the application is presented and in section 4.6 the numerical simulations of the heat exchanger in 3D is discussed.

4.1 Heat Equation Application Testing

In this section, a comparison between the numerical solution and an analytical solution for a specific problem has been made. For the next few cases a unit square domain, $\Omega = [0.1]\text{x}[0, 1]$ has been tested.

- pure diffusion
- diffusion dominant
- convection dominant
- ASGS stablization

For each problem dimensionless properties have been considered as ρ=1, c_p=1 and k=1 and the time discretization scheme used has been BDF2.

4.1.1 Pure Diffusion

The pure diffusion case, considering the heat equation (Eq. 2.14), is obtained setting the convective velocity **a** to zero.

The problem in differential form is:

$$\rho c_p \frac{\partial T}{\partial t} - \nabla \cdot (k \nabla T) = q \quad in \quad \Omega \tag{4.1}$$

The pure diffusion case is defined by the following equations:

$$\frac{\partial T}{\partial t} - \nabla \cdot (\nabla T) = 4xy \quad in \quad \Omega \tag{4.2a}$$

$$T(x, y, 0) = x^2 + y^2 \quad in \quad \Omega \tag{4.2b}$$

$$T(x, y, t) = x^2 + y^2 + xyt \quad on \quad \partial\Omega \tag{4.2c}$$

and the source term has been computed as:

$$q = 4xy$$

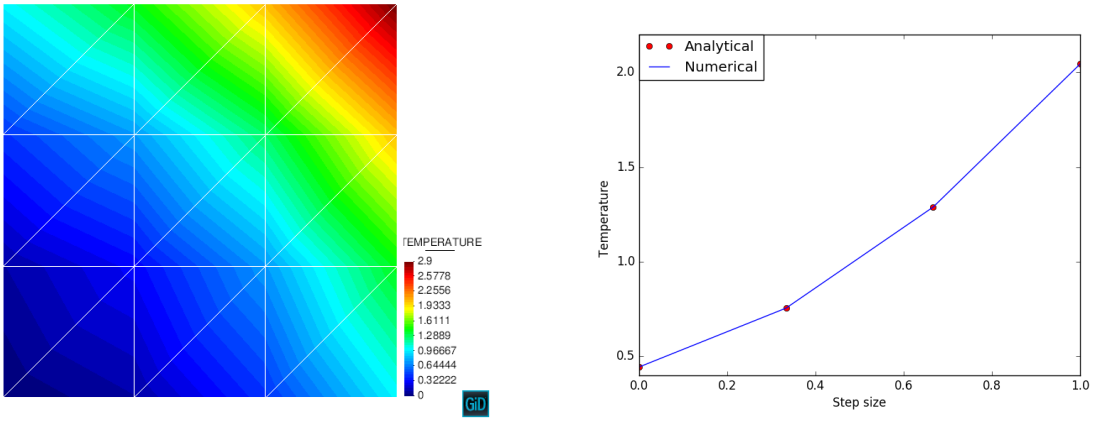
The exact solution of the problem is:

$$T(x, y, t) = x^2 + y^2 + xyt$$

The numerical solution has been computed with a structured mesh composed by 18 uniform triangular linear element.

Figure 4.1(a) shows the temperature solution of a square domain and Figure 4.1(b) represents the comparison between the numerical and the analytical solution.

The results show how the computational results obtained match perfectly with analytical solution.



((a)) Temperature distribution.

((b)) Analytical solution vs numerical solution.

Figure 4.1: Results of a pure diffusion case at dimensionless time t = 1.

4.1.2 Diffusion Dominant

The diffusion dominant problem has been tested by computing a structured mesh composed of 400 uniform linear triangular element and taking the convective velocity as:

$$\mathbf{a} = (150, 0)$$

where the notation for the convective velocity is $\mathbf{a} = (a_x, a_y)$.

Considering the element size h equal to 0.05, the Peclet number is 7.5.

The differential equation is defined by the following equations:

$$\frac{\partial T}{\partial t} + \mathbf{a} \cdot (\nabla T) - \nabla \cdot (\nabla T) = q \quad in \quad \Omega \tag{4.3a}$$

$$T(x, y, 0) = a_x(x^2 + y^2) \quad in \quad \Omega \tag{4.3b}$$

$$T(x, y, t) = a_x(x^2 + y^2 + xyt) \quad on \quad \partial\Omega \tag{4.3c}$$

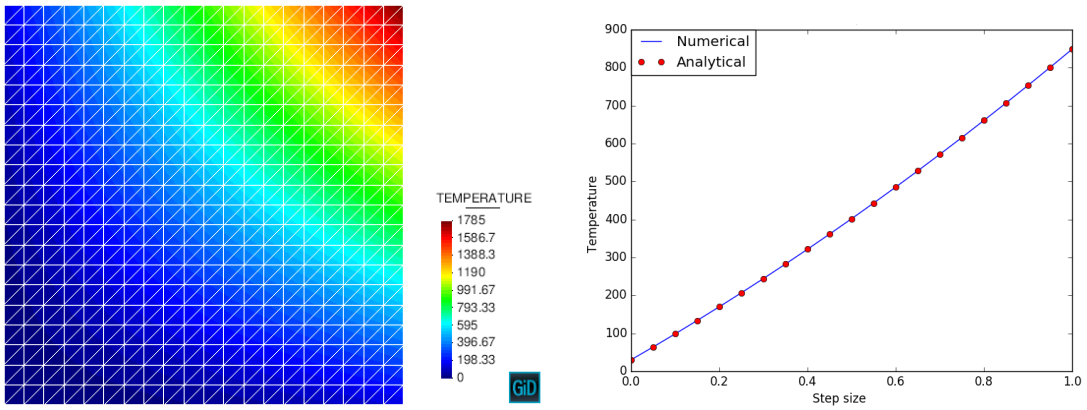
The source term has been computed as:

$$q = a_x(xy + a_x(2x + yt) - 4)$$

The exact solution of the problem is:

$$T(x, y, t) = a_x(x^2 + y^2 + xyt)$$

Figure 4.2(b) shows how also for a convective term different than zero, and for a Peclet number Pe = 7.5, the numerical solution matches perfectly with the exact solution.



((a)) Temperature distribution.

((b)) Analytical solution vs numerical solution.

Figure 4.2: Results of a diffusion dominant case for Pe = 7.5 at dimensionless time t = 10.

4.1.3 Convection Dominant

The convection dominant problem has been tested using the same structured mesh generated for the previous case and taking the convective velocity as:

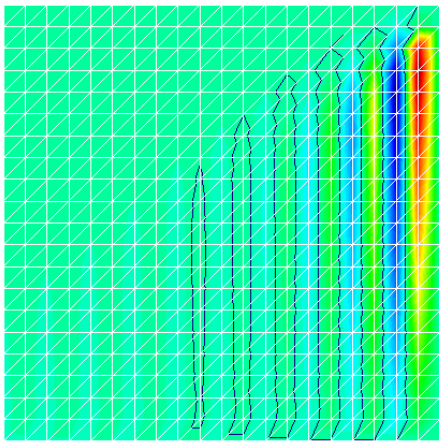
$$\mathbf{a} = (200, 0)$$

Considering the element size h equal to 0.05, the Peclet number is 10.

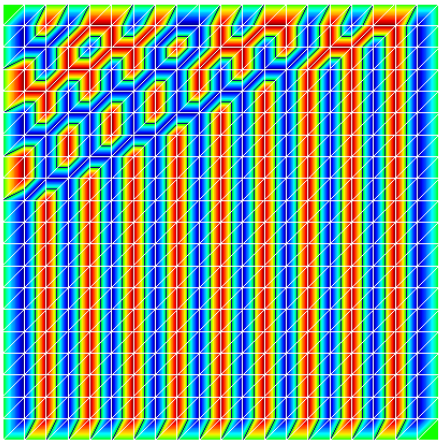
Results demonstrate that Galerkin method is not ideally suited to solve convection-dominated problems. The Galerkin method loses its best approximation property when the non-symmetric convective operator dominates the diffusive operator in the transport equation, and consequently spurious node-to-node oscillations appear [Huerta].

It is clear that for high values of the Peclet number the Galerkin method fails. This instability can be overcome by a highly meshed refinement but unfortunately, due to the computationally cost of the simulation, is not a viable approach.

Figure 4.3(a) and Figure 4.3(b) show an unstable behaviour of the solution. In order to obtain better results a stabilized formulation must be implemented toward matching the exact solution at each node of a uniform mesh of linear elements for any mesh size h and all values of the Peclet number.



((a)) Solution at dimensionless time t = 0.3.



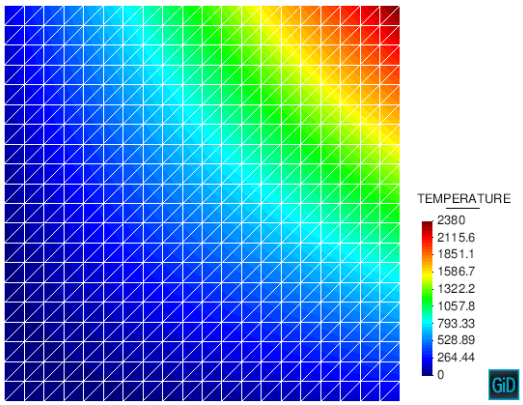
((b)) Solution at dimensionless time t = 10.

Figure 4.3: Results of a convection dominant case for Pe = 10 for different time steps.

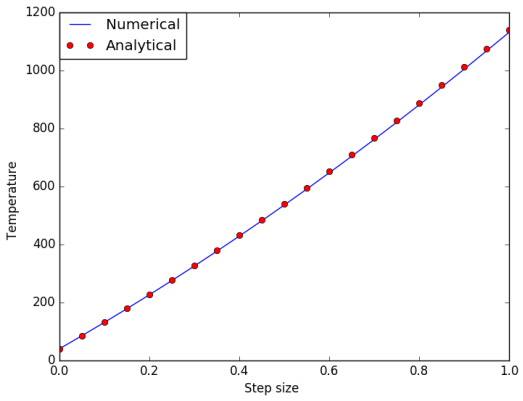
4.1.4 ASGS implementation

In order to avoid the instabilities observed in the convective dominant problem, the ASGS stabilization technique has been added to the finite element formulation considering the same problem of the previous section 4.1.3.

After the ASGS technique has been implemented in Kratos as stated in section 2.3.1, the result clearly shows how the stability has been reached with perfect matching between the analytical and the numerical solution for a Peclet number equal to 10 (Figure 4.4).



((a)) Temperature distribution.



((b)) Analytical solution vs numerical solution.

Figure 4.4: Results of a convection dominant case for Pe = 10 at dimensionless time t = 10.

4.2 Neumann Flux Condition Testing 2D

In this section, in order to test the flux condition, the Poisson equation is solved using Neumann and Dirichlet boundary conditions and the numerical results are compared with a typical case of study *Heat conduction through a large plane wall* (see [11]).

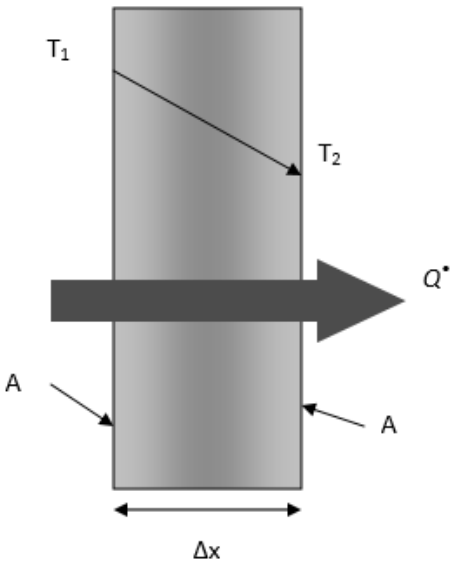


Figure 4.5: Heat conduction through a large plane wall (Cengel).

For the next case a rectangular domain, $\Omega = [0,1]$x$[0,4]$ has been tested. The problem reads:

$$-\Delta T = f \quad in \quad \Omega \tag{4.4a}$$

$$\nabla T \cdot n = \dot{Q} \quad on \quad \partial\Omega_{left} \tag{4.4b}$$

$$\nabla T \cdot n = -\dot{Q} \quad on \quad \partial\Omega_{right} \tag{4.4c}$$

$$T_1 = h \quad on \quad \partial\Omega_L \tag{4.4d}$$

where $\dot{Q} = 4010[W/m^2]$ and $h = 303[K]$.

The results show the temperature T_2 achieved on $\partial\Omega_{right}$ when the system reached stability corresponds to 293 K, which is exactly as the analytical solution provided by the bibliography.

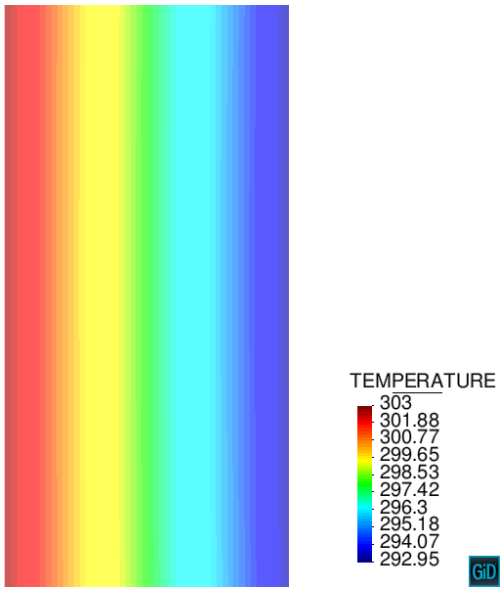


Figure 4.6: Results of a pure diffusion case imposing flux condition through a large plane wall.

4.3 CFD-Heat Transfer Application Testing 2D

4.3.1 Laminar Flow Around a Cylinder

For the CFD-Heat Transfer Application the 2D classical test case simulating the fluid flow ($\Omega 1$) around a cylinder with a circular cross section ($\Omega 2$) has been considered. In the current problem the cylinder domain is considered empty and only the fluid domain is taken into account.

The fluid in question is an incompressible Newtonian fluid where the kinematic viscosity $\nu = 10^{-3} m^2/s$, the fluid density is $\rho = 1\ kg/m^3$, the thermal conductivity $k = 0.0257\ W/(mK)$ and the specific heat is $c_p = 1.005\ kJ/(kgK)$.

The entire domain ($\Omega = \Omega 1 \cup \Omega 2$) is described in Figure 4.7, where H is the height of the channel and D is the cylinder diameter.

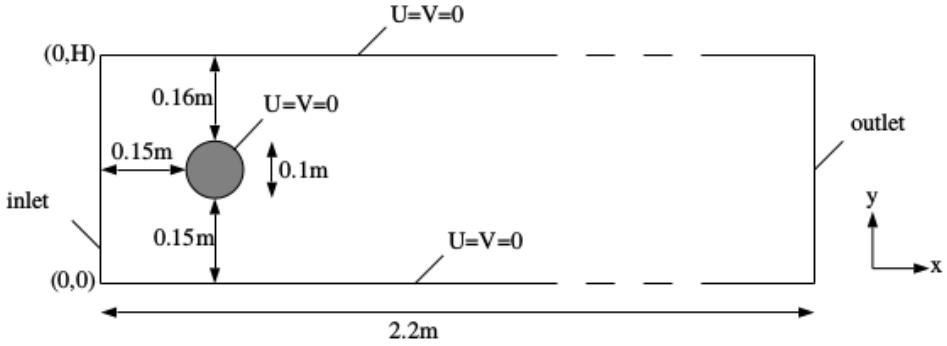


Figure 4.7: Geometry of 2D test cases with boundary conditions (Schäfer).

The problem is governed by the following set of differential equations and initial and boundary conditions (Eq. 4.5). The mechanical part of the problem is governed by the Navier-Stokes equations (Eq. 4.5a and Eq. 4.5b) and the thermal part of the problem is governed by the heat equation (Eq. 4.5c):

$$\nabla \cdot \mathbf{a} = 0 \quad in \quad \Omega 1 \tag{4.5a}$$

$$\frac{\partial \mathbf{a}}{\partial t} + (\mathbf{a} \cdot \nabla)\mathbf{a} - \nu \nabla^2 \mathbf{a} = -\frac{1}{\rho}\nabla p + \mathbf{g} \quad in \quad \Omega 1 \tag{4.5b}$$

$$\rho c_p \frac{\partial T}{\partial t} + \rho c_p \mathbf{a} \cdot (\nabla T) - \nabla \cdot (\nabla T) = 0 \quad in \quad \Omega \tag{4.5c}$$

$$T(x, y, 0) = 298.15 \quad in \quad \Omega 1 \tag{4.5d}$$

$$T(0, y, t) = 298.15 \quad at \quad \partial\Omega 1_{INLET} \tag{4.5e}$$

$$T(x, y, 0) = 355.15 \quad at \quad \partial\Omega 2 \tag{4.5f}$$

$$U(0, y, t) = 4U_m y(H - y)/H^2, V = 0 \quad at \quad \partial\Omega 1_{INLET} \tag{4.5g}$$

$$V(L, y, t) = 0 \quad at \quad \partial\Omega 1_{OUTLET} \tag{4.5h}$$

The *inlet* velocity condition (Eq. 4.5g) describe a parabolic profile which depends on the mean velocity

$$U_m(t) = 2U(0, H/2, t)/3$$

with $U_m = 1.5\ m/s$, yielding the Reynold number $Re = 100$.

At the wall and around the cylinder the no-slip condition have been assigned.

For the *outlet* condition (Eq. 4.5h) the vertical velocity component at the end of the domain (L = 2.2 m) has been set to zero.

The temperature initial condition define for all the fluid a temperature equal to 298.15 K (Eq. 4.5d) and the cylinder temperature equal to 355.15 K (Eq. 4.5f).

A fixed temperature condition , T = 298.15 K, has been set also for the fluid the at the inlet (Eq.4.5).

The time step simulation has been set to 0.05 for a total of 100 steps.

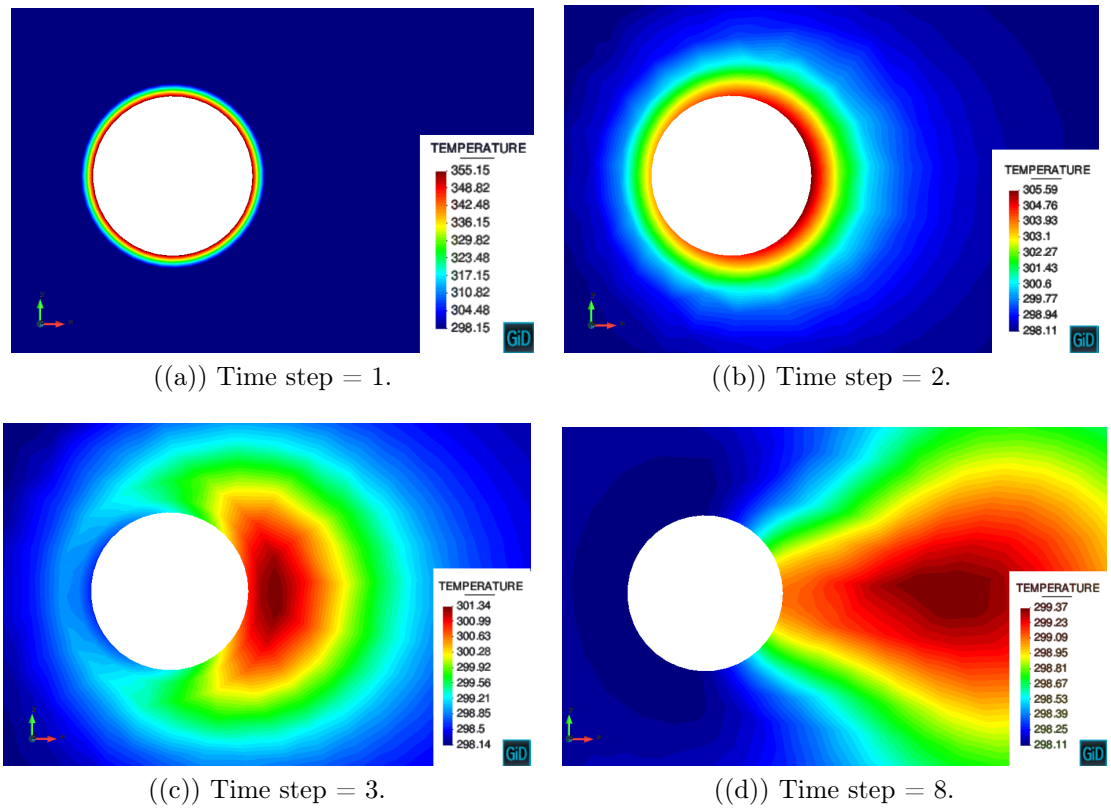
Results for different time steps show how the heat transfer between the fluid and the heated circumference has been achieved through convection.

Figure 4.8(a) shows the initial temperature condition for the circumference and the fluid. Here, the difference of temperature between the two domains is clearly visible , where the cylinder is heated up to 355.15 K and the fluid, still static, has a temperature of 298.15 K.

At the second time step (4.8(b)) how the temperature decreases in the whole domain can be seen . After 0.05 s the maximum temperature registered decreased

down to 305.59 K. It is also visible how the fluid has been warmed up near the cylinder and how the heat is transported by the flow.

This behaviour can be better appreciated on Figure 4.8(d) and Figure 4.8(b) where clearly the circumference cools down and almost all the heat has been transferred to the fluid and transported to the outlet. After 0.4 s, the maximum temperature decreased down to 299.37 K.



((a)) Time step = 1.

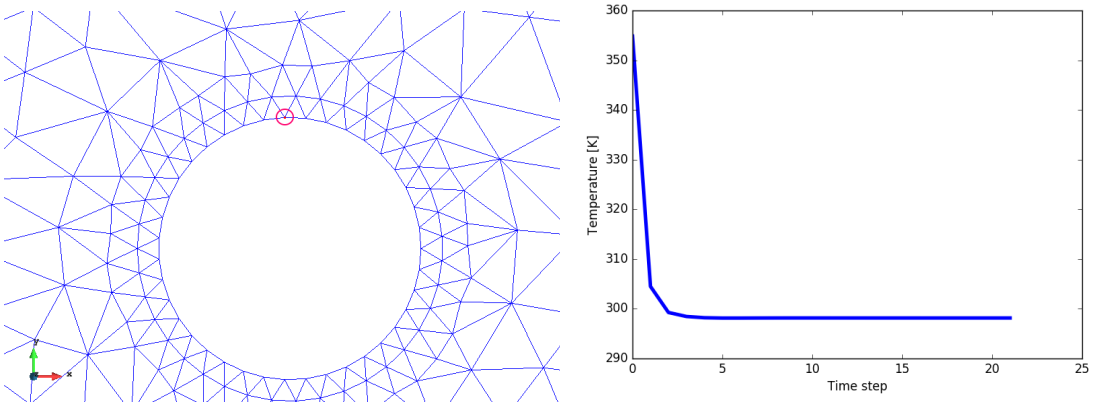
((b)) Time step = 2.

((c)) Time step = 3.

((d)) Time step = 8.

Figure 4.8: Temperature field results at different time steps of a laminar flow around a cylinder.

The temperature evolution of a point on the circumference (Figure 4.9) shows how the temperature drops down during the first 2 time steps and then becomes constant approximately reaching 298 K after 5 time steps.



((a)) Point under study localized around the cylinder.

((b)) Plot time steps vs temperature.

Figure 4.9: Temperature evolution of a point located on the cylinder.

4.3.2 Laminar Flow Around a Heated Cylinder

After testing the CFD-heat transfer application the next step has been to test a problem where two different domain are involved. In order to compute the evolution of the temperature in the domain $\Omega 2$, a solid material has been assigned. The fluid's data for the domain $\Omega 1$ is the same as in the previous section 4.3.1.

The material assigned to the cylinder has a thermal conductivity $k = 1000$ $W/(mK)$, the specific heat is $c_p = 0.1$ $kJ/(kgK)$ and the density is $\rho = 8940$ kg/m^3.

The problem is governed by the following set of differential equations and initial and boundary conditions (Eq.4.6):

$$\nabla \cdot \mathbf{a} = 0 \quad in \quad \Omega 1 \tag{4.6a}$$

$$\frac{\partial \mathbf{a}}{\partial t} + (\mathbf{a} \cdot \nabla)\mathbf{a} - \nu \nabla^2 \mathbf{a} = -\frac{1}{\rho}\nabla p + \mathbf{g} \quad in \quad \Omega 1 \tag{4.6b}$$

$$\rho c_p \frac{\partial T}{\partial t} + \rho c_p \mathbf{a} \cdot (\nabla T) - \nabla \cdot (\nabla T) = 0 \quad in \quad \Omega \tag{4.6c}$$

$$T(x, y, 0) = 298.15 \quad in \quad \Omega 1 \tag{4.6d}$$

$$T(0, y, t) = 298.15 \quad at \quad \partial\Omega 1_{INLET} \tag{4.6e}$$

$$T(x, y, 0) = 355.15 \quad at \quad \Omega 2 \tag{4.6f}$$

$$U(0, y, t < 5) = 4yU_m sin(0.1\pi t)(H - y)/H^2, V = 0 \quad at \quad \partial\Omega 1_{INLET} \tag{4.6g}$$

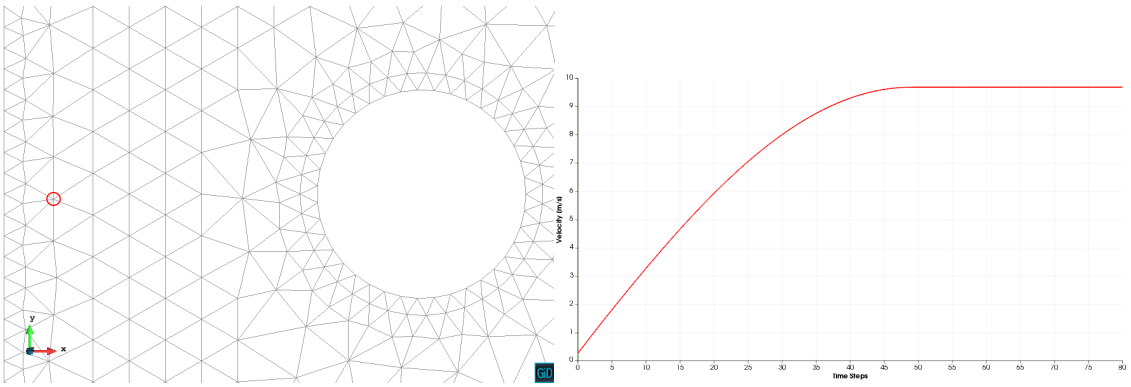
$$U(0, y, t \geq 5) = 4yU_m(H - y)/H^2, V = 0 \quad at \quad \partial\Omega 1_{INLET} \tag{4.6h}$$

$$V(L, y, t) = 0 \quad at \quad \partial\Omega 1_{OUTLET} \tag{4.6i}$$

In order to have a more effective heat transfer the value of the velocity at the inlet has been increased from zero to an avarage velocity of 5 m/s in the time interval $0 \leq t < 5$ s (Eq. 4.6g). This gives a time varying Reynolds number between $0 \leq Re(t) < 4000$. After 5 s the velocity remains constant until the end of the simulation (Eq. 4.6h).

The time step simulation has been set to 0.1 for a total of 80 steps.

Figure 4.10(b) shows how the flow velocity starts from zero and after 5 seconds (50 time steps) it gets a stable velocity almost of 9.5 m/s.



((a)) Point under study localized at the inlet .

((b)) Plot time steps vs velocity.

Figure 4.10: Transient velocity behaviour at the inlet .

In this case the two domains have matching nodes at the interface (Figure 4.11).

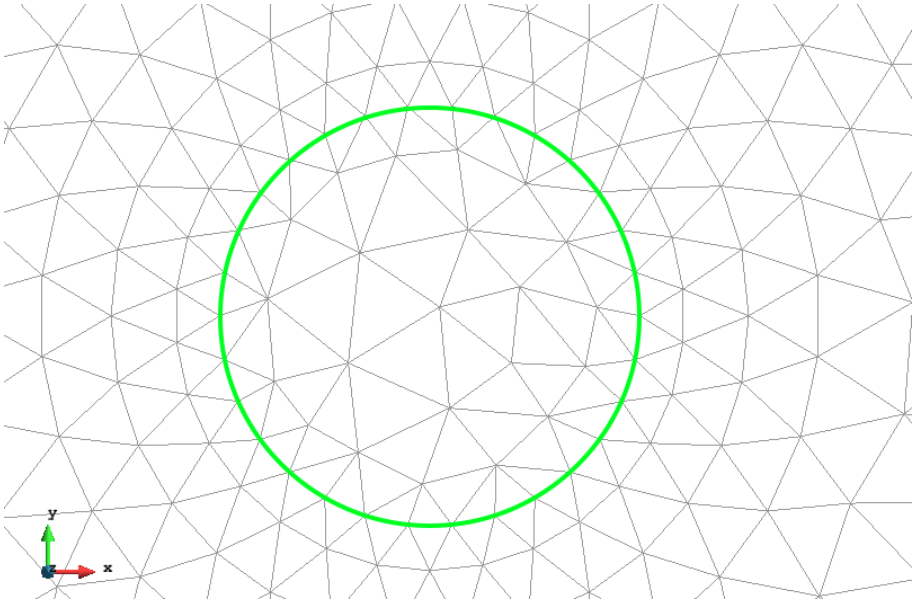
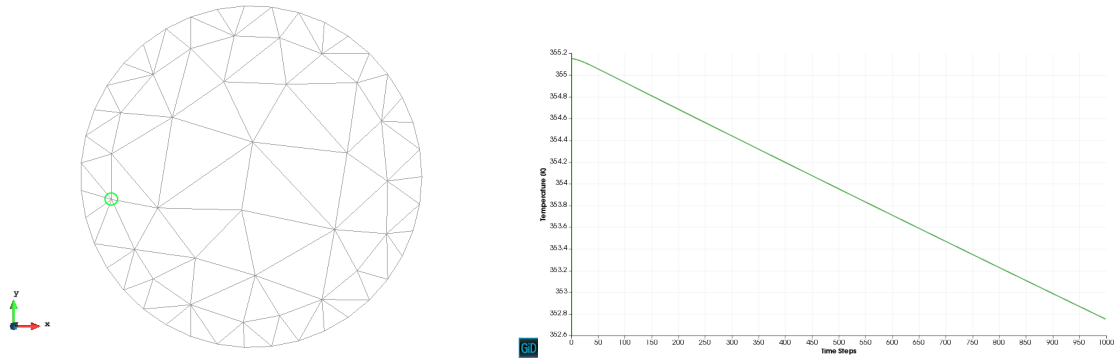


Figure 4.11: Matching nodes between the solid and the fluid domain at the interface (green circle).

Temperature results for the solid domain (Figure 4.12) shows that the temperature has decreased in time so the communication between domains has been effective. Specifically, for a point inside the solid domain, the temperature has decreased from 355.18 K down to 352.6 K in 100 time steps.

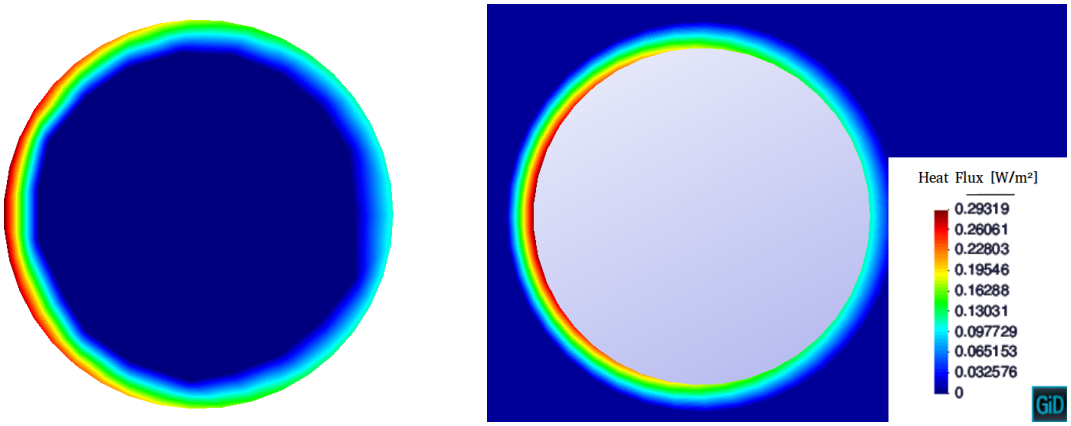


((a)) Point under study.

((b)) Plot time steps vs temperature.

Figure 4.12: Temperature evolution for a node localized at the interior of the solid .

Figure 4.13 shows the value of the fluxes at time t = 0.3 s for both domains. As can be appreciated the fluxes at the interface are the same due to the imposition of the fluid's reactions as Neumann boundary conditions to the solid domain. It is also visible how the value of the flux is greater at the cylinder's side facing the inlet. This result can be explained due to the temperature difference between the flow and the heated solid is greater and that generates bigger fluxes compared with the cylinder side facing the outlet.



((a)) Solid domain. ((b)) Fluid domain.

Figure 4.13: Heat flux results at time = 0.3 s.

4.4 Heat Exchanger's Numerical Simulation in 2D

In this chapter the 2D heat exchanger model has been simulated for a turbulent case. During the simulation it has not been taking into account the presence of all the cylinders but only one of them heated. The heated cylinder is located at the first cylinder's row and the no heated cylinders are considered as an empty space with no material properties (Figure 4.15)).

4.4.1 Description of the Problem

The governing equations of the problem reads:

$$\nabla \cdot \mathbf{a} = 0 \quad in \quad \Omega 1 \tag{4.7a}$$

$$\frac{\partial \mathbf{a}}{\partial t} + (\mathbf{a} \cdot \nabla)\mathbf{a} - \nu \nabla^2 \mathbf{a} = -\frac{1}{\rho}\nabla p + \mathbf{g} \quad in \quad \Omega 1 \tag{4.7b}$$

$$\rho c_p \frac{\partial T}{\partial t} + \rho c_p \mathbf{a} \cdot (\nabla T) - \nabla \cdot (\nabla T) = 0 \quad in \quad \Omega \tag{4.7c}$$

$$T(x, y, 0) = 298.15 \quad in \quad \Omega 1 \tag{4.7d}$$

$$T(0, y, t) = 298.15 \quad at \quad \partial \Omega 1_{INLET} \tag{4.7e}$$

$$T(x, y, 0) = 355.15 \quad at \quad \Omega 2 \tag{4.7f}$$

$$U(0, y, t < 1) = 4yU_m sin(0.1\pi t)(H - y)/H^2, V = 0 \quad at \quad \partial \Omega 1_{INLET} \tag{4.7g}$$

$$U(0, y, t > 1) = 4yU_m (H - y)/H^2, V = 0 \quad at \quad \partial \Omega 1_{INLET} \tag{4.7h}$$

$$V(L, y, t) = 0 \quad at \quad \partial \Omega 1_{OUTLET} \tag{4.7i}$$

$$P(L, y, t) = 0 \quad at \quad \partial \Omega 1_{OUTLET} \tag{4.7j}$$

where $\Omega 1$ is the body of the heat exchanger domain and $\Omega 2$ is the heated cylinder domain.

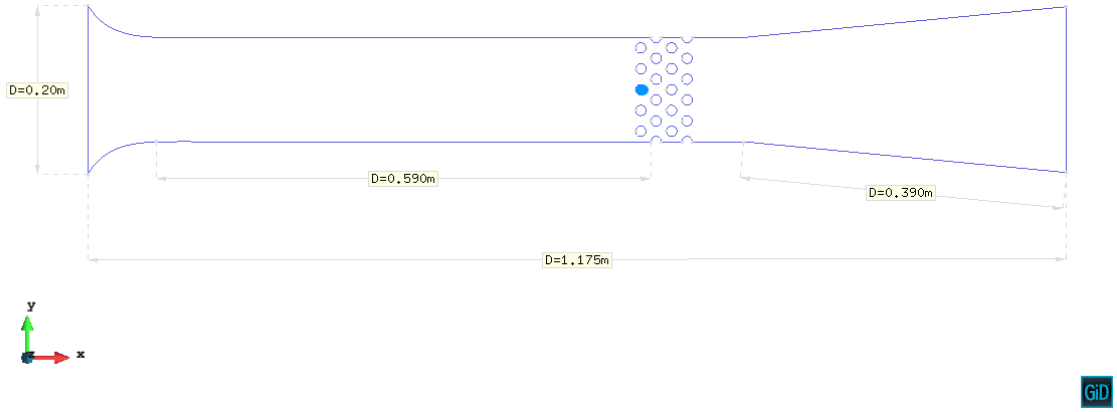


Figure 4.14: Geometry and dimensions of the heat exchanger.

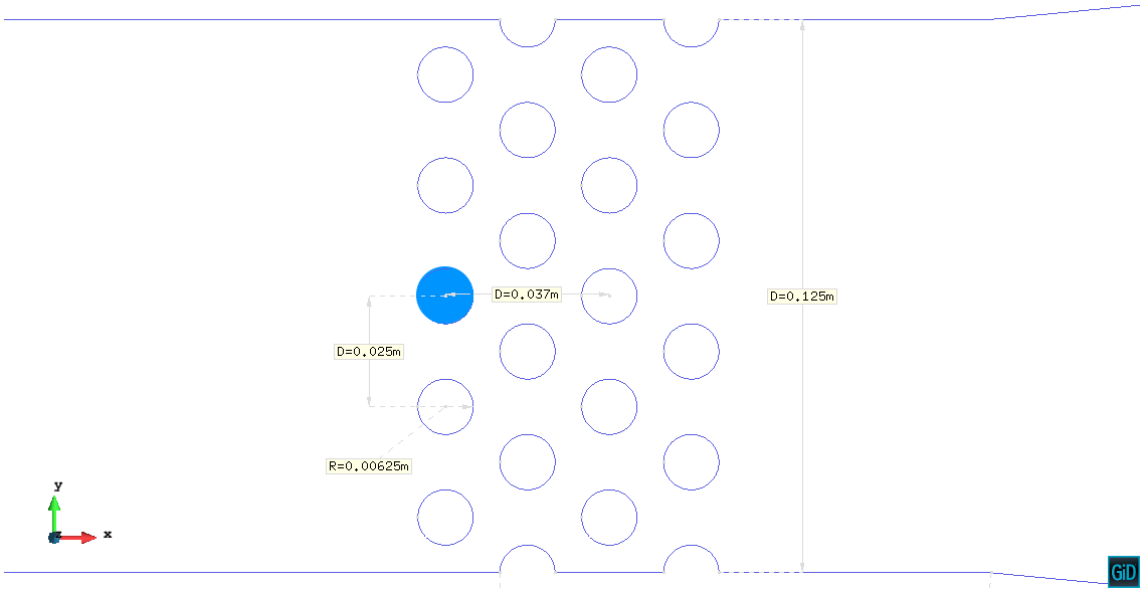


Figure 4.15: Geometry and dimensions of the heat exchanger's test area.

The problem involves an incompressible flow entering from the heat exchanger's nozzle (inlet) describing a parabolic profile depending on the mean velocity U_m. The fluid initial temperature is $T = 298.15$ K and the solid initial temperature is $T = 353.15$ K. Also at the inlet the temperature flow has been fixed to 298.15 K during all the simulation. At the outlet the vertical velocity V and the pressure P

have been set equal to zero. The *no-slip* condition has been applied around all the cylinders and at the heat exchanger's walls.

The fluid in question is an incompressible Newtonian fluid where the kinematic viscosity ν= $1.47760E^{-5}$ m^2/s, the fluid density is ρ= 1.205 kg/m^3, the thermal conductivity k= 0.0257 $W/(mK)$ and the specific heat c_p= 1.005 $kJ/(kgK)$. For the solid the density is ρ= 8940 kg/m^3, the thermal conductivity k= 401 $W/(mK)$ and the specific heat c_p= 0.39 $kJ/(kgK)$.

4.4.2 Velocity Definition

Due to the fact that the geometry of the heat exchanger varies along its length, velocity 's changes are taken into account.

Continuity equation establishes

$$\rho_1 A_1 U_1 = \rho_2 A_2 U_2$$

where U_1 is the horizontal component of the inlet velocity, U_2 is the horizontal velocity between the cylinders at the test area, A_1 is the inlet area and A_2 is the area between the first row of cylinders.

Due to the flow is incompressible and considering the 2D case, the equation can be rewritten as.

$$h_1 U_1 = h_2 U_2$$

where h_1 = 0.2 m and h_2 = 0.0625 m.

Depending on the inlet velocity, the maximum velocity U_2 is obtained as:

$$U_2 = 0.2\frac{U_1}{0.0625} \tag{4.8}$$

4.4.3 Mesh Size Definition

As the the Reynolds number increases it is important that the mesh is properly sized to ensure an accurate simulation near the walls. The mesh element size depends on the dimension of the small vortexes, where the kinetic energy is converted into heat energy due to the viscosity effects.

In order to find an appropriate element size near the walls the mesh will be very dense, no approximation is needed and the solution will be more accurate.

The mesh domains, both for $\Omega 1$ and for $\Omega 2$, have been generated using a triangular unstructured mesh adopting a reference element size h_{max} = 0.008. Near

the cylinders and near the walls, the element size has been taken respectively as $h_{interface} = 0.0001$ and $h_{walls} = 0.000337$.

The number of triangular elements generated for $\Omega 1$ is 37.391 and for $\Omega 2$ is 523.

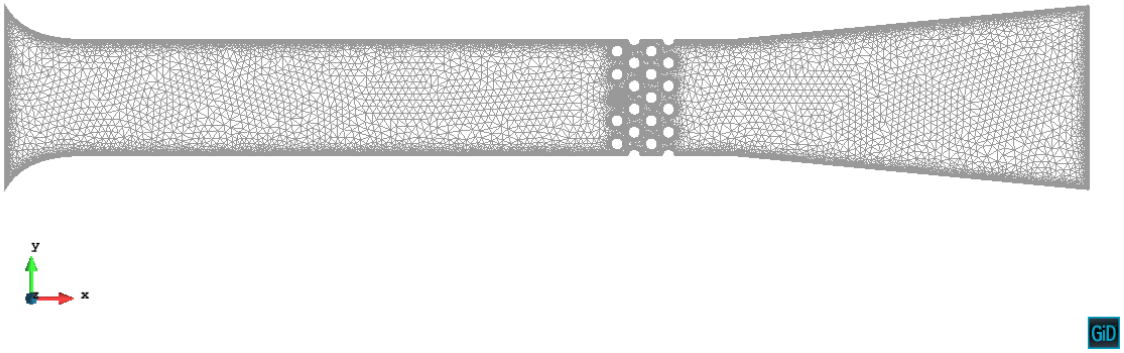


Figure 4.16: Triangular unstructured mesh of the whole domain.

Figure 4.17 shows a different number of elements density near the cylinders and near the walls due to the importance to obtain a really accurate resolution of the flow field.

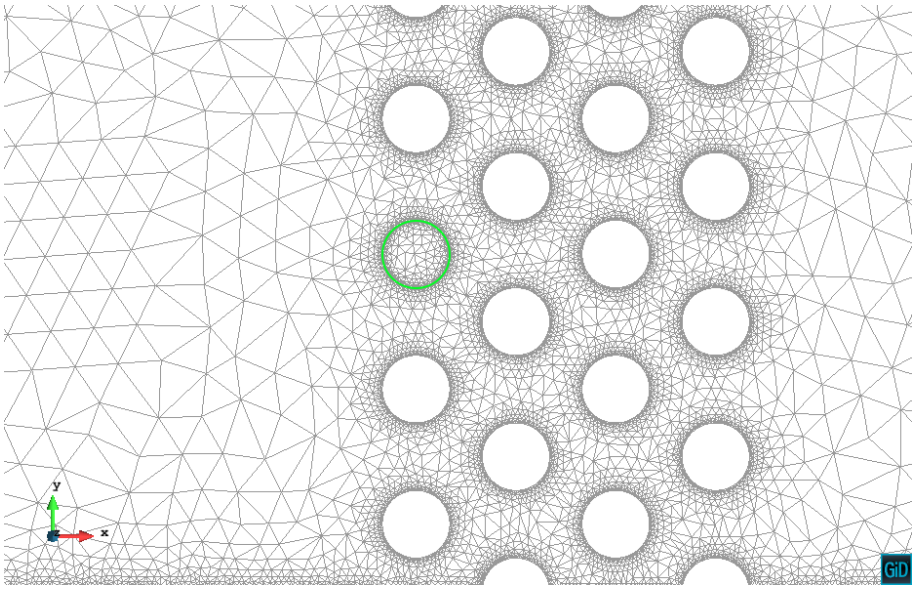


Figure 4.17: Very dense mesh in the tube banks area.

Figure 4.18 shows that the two domains' mashes have matching nodes at the interface in the heat exchanger test area.

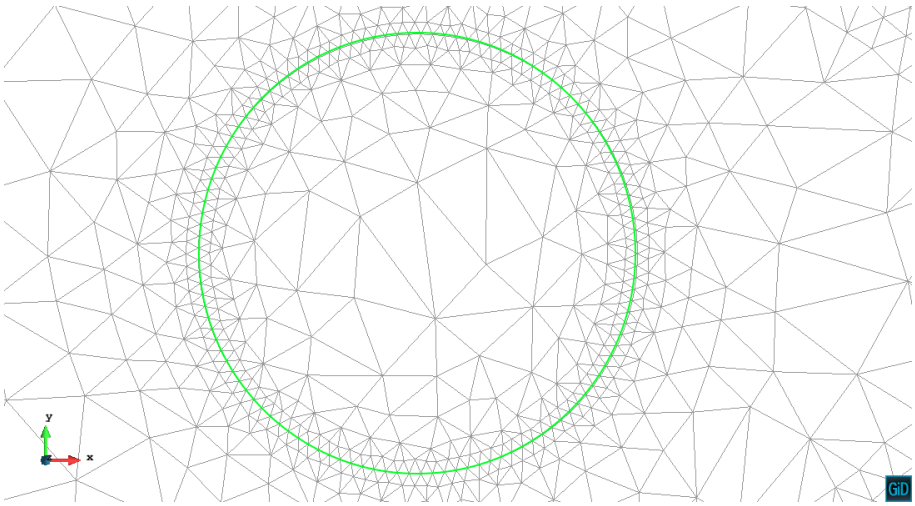


Figure 4.18: Two domains interface matching nodes (green circle).

4.4.4 Fluid Dynamic Results

Figure 4.19 shows how the velocity increases as the flux gets into the nozzle, before reaching the tube banks, creating a profile not properly symmetric because the turbulent flux is not yet fully developed (Figure 4.20). As the flux gets through the tube banks the velocity increases up to a maximum value of 18.677 m/s and as it gets to the diffuser area, the velocity flux decreases and fluctuation starts to appear.

For both profiles, before and after the tube banks, it can be seen how the no sleep condition has been fulfilled near the walls.

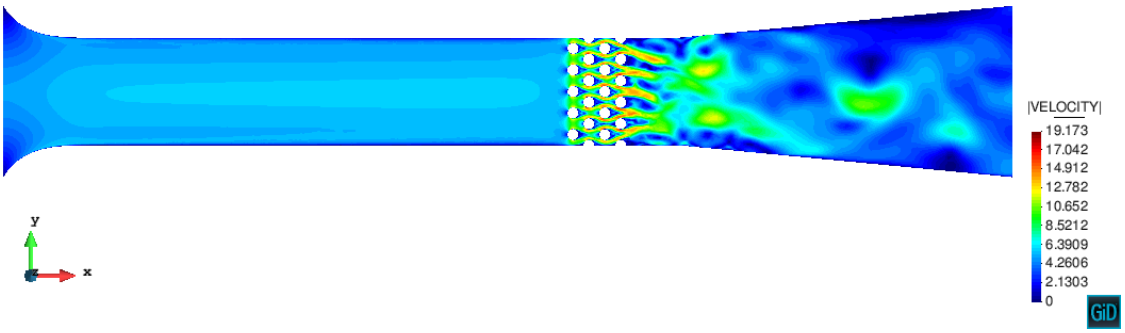


Figure 4.19: Velocity distribution at 1 seconds.

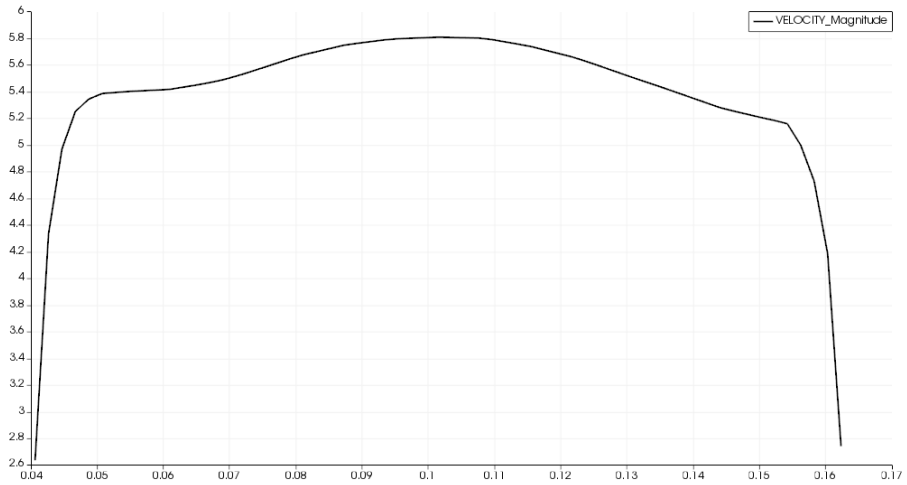


Figure 4.20: Velocity profile before the tube banks.

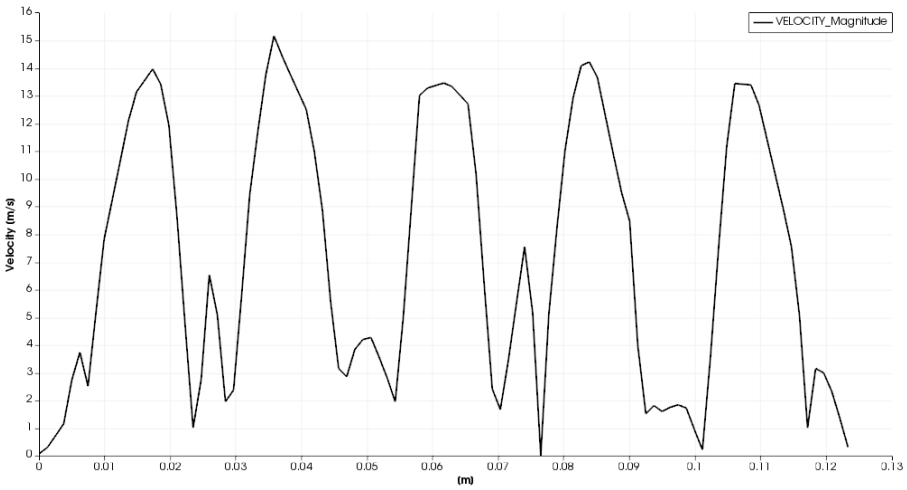


Figure 4.21: Velocity profile after the tube banks.

4.4.5 Heat Transfer Results

Results related with change in temperature are shown next. The temperature changes related to the air are observed in Figure 4.22 meanwhile Figure 4.23 shows the time temperature evolution of a point taken on the cylinder surface. The cylinder temperature decreased uniformly more than 2 K for its whole surface during the first second of simulation.

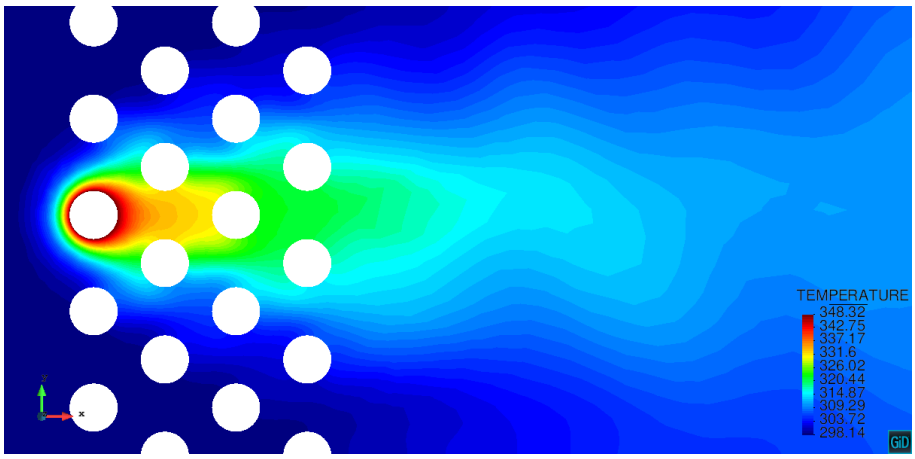
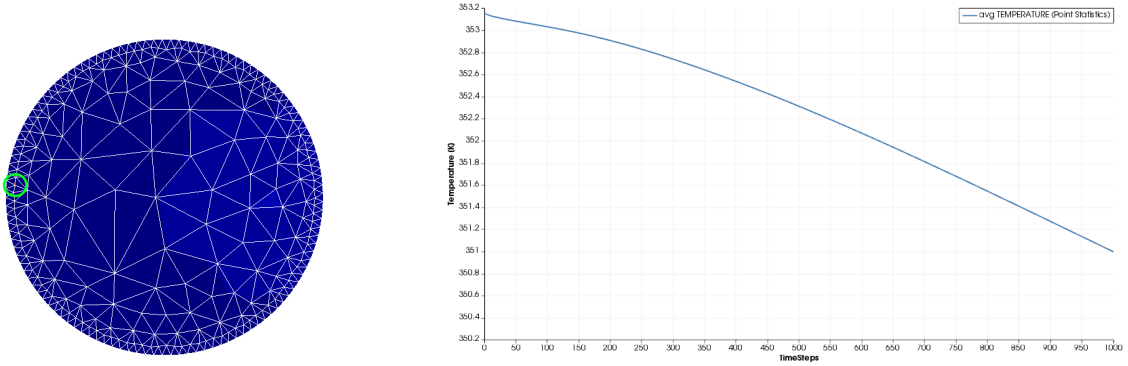


Figure 4.22: Temperature distribution after 1 seconds.



((a)) Point over the cylinder under study.

((b)) Temperature evolution.

Figure 4.23: Cylinder temperature evolution after 1 second.

It is important to point out that due to the fact that turbulence is a phenomenon involving three-dimensional vorticity fluctuations, the results obtained are not reliable unless a turbulent model is used.

In order to obtain more consistent results, the 3D model has been studied and it is presented in the next chapter.

4.5 Flow Around a Heated Cylinder 3D

In order to test the CFD-Heat Transfer Application in 3D the typical case flow around a cylinder, has been tested.

4.5.1 Description of the Problem

Also for this case the fluid in question is an incompressible Newtonian fluid where the kinematic viscosity $\nu = 10^{-3}m^2/s$, the fluid density is $\rho = 1\ kg/m^3$, the thermal conductivity $k = 0.0257\ W/(mK)$ and the specific heat is $c_p = 1.005\ kJ/(kgK)$.

The entire domain ($\Omega = \Omega 1 \cup \Omega 2$) is described in Figure 4.24, where H is the height of the channel and D is the cylinder diameter.

The material assigned to the cylinder has a thermal conductivity $k = 1000$ $W/(mK)$, the specific heat is $c_p = 0.1\ kJ/(kgK)$ and the density is $\rho = 8940\ kg/m^3$.

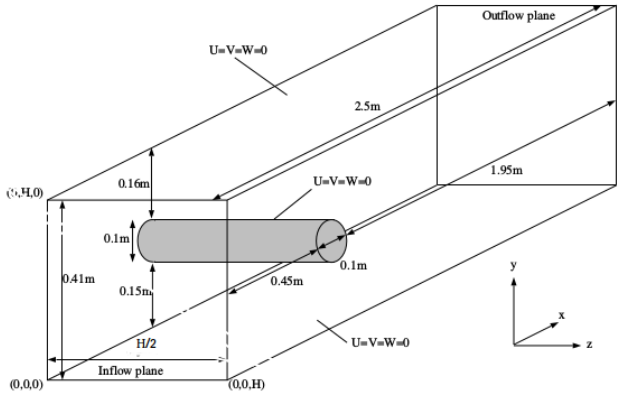


Figure 4.24: Configuration for flow around a cylinder with circular cross–section (Schafer).

The problem is governed by the following set of differential equations and initial and boundary conditions (Eq.4.9):

$$\nabla \cdot \mathbf{a} = 0 \quad in \quad \Omega 1 \tag{4.9a}$$

$$\frac{\partial \mathbf{a}}{\partial t} + (\mathbf{a} \cdot \nabla)\mathbf{a} - \nu \nabla^2 \mathbf{a} = -\frac{1}{\rho}\nabla p + \mathbf{g} \quad in \quad \Omega 1 \tag{4.9b}$$

$$\rho c_p \frac{\partial T}{\partial t} + \rho c_p \mathbf{a} \cdot (\nabla T) - \nabla \cdot (\nabla T) = 0 \quad in \quad \Omega \tag{4.9c}$$

$$T(x, y, z, 0) = 298.15 \quad in \quad \Omega 1 \tag{4.9d}$$

$$T(0, y, z, t) = 298.15 \quad at \quad \partial\Omega 1_{INLET} \tag{4.9e}$$

$$T(x, y, z, t > 5) = 355.15 \quad at \quad \Omega 2 \tag{4.9f}$$

$$U(0, y, z, t < 5) = 16U_m yz(H-y)(H/2-z)sin(C\frac{\pi}{2}t)/(H^4/4), V = 0, Z = 0 \quad at \quad \partial\Omega 1_{INLET} \tag{4.9g}$$

$$U(0, y, z, t \geq 5) = 16U_m yz(H-y)(H/2-z)/(H^4/4), V = 0, Z = 0 \quad at \quad \partial\Omega 1_{INLET} \tag{4.9h}$$

$$V(L, y, z, t) = 0 \quad at \quad \partial\Omega 1_{OUTLET} \tag{4.9i}$$

$$P(L, y, z, t) = 0 \quad at \quad \partial\Omega 1_{OUTLET} \tag{4.9j}$$

where $\Omega 1$ is the body of the heat exchanger domain, $\Omega 2$ is the heated cylinder domain and C is a constant depending on the time step.

4.5.2 Mesh Size Definition

The reference element size chosen has been $h_{max} = 0.01$ and near the walls and the cylinders the element size has been taken respectively as $h_{min} = 0.001$. Once the mesh has been generated, the cylinder's mesh was composed by 1.232 triangular elements and 3.783 tetrahedral elements and the fluid's mesh by 15.460 triangular elements and 199.595 tetrahedral elements. Also for the 3D case at the interface between the cylinder and the fluid, the two domains have matching nodes.

4.5.3 Results

Figure 4.25 and Figure 4.26 show how the heat transfer has been effective also for he 3D case. Results show the temperature fields for both the fluid and solid domains. After 6 seconds the solid temperature has decreased from 353.15 K down to 352.09 K (Figure 4.27).

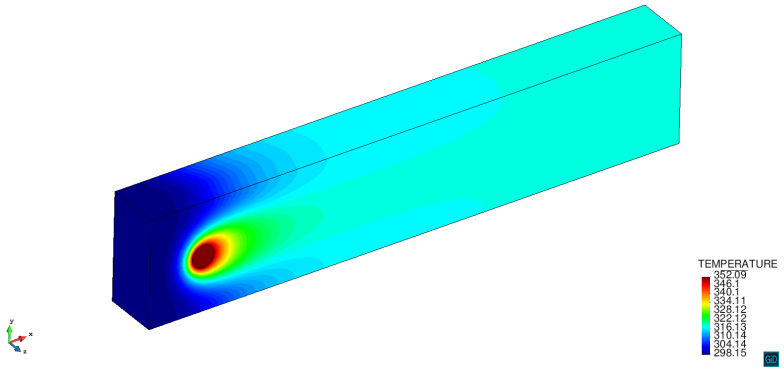


Figure 4.25: Fluid's temperature field after 6 seconds.

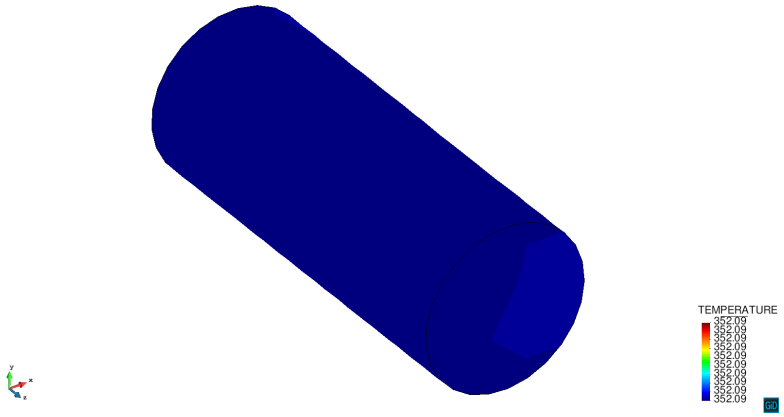
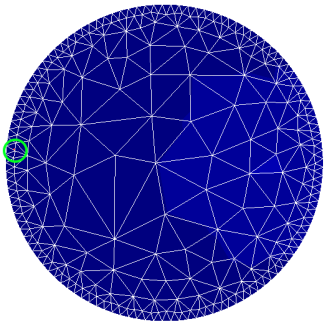
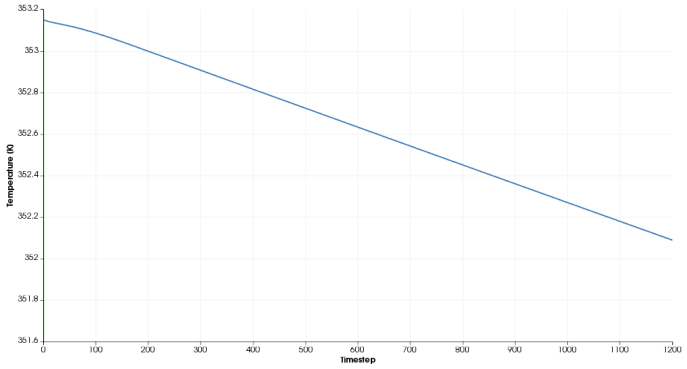


Figure 4.26: Solid's temperature field after 6 seconds.



((a)) Point over the cylinder under study.



((b)) Temperature evolution.

Figure 4.27: Cylinder temperature decreasing evolution during 6 seconds.

The next step has been to test the 3D heat exchanger model and compare the numerical results with the experimental data.

4.6 Heat Exchanger Numerical Simulation in 3D

Once the problem in 2D has been solved the next step has been creating a 3D heat exchanger model using the same 2D model's conditions.

4.6.1 Description of the Problem

The governing equations of the problem reads:

$$\nabla \cdot \mathbf{a} = 0 \quad in \quad \Omega 1 \tag{4.10a}$$

$$\frac{\partial \mathbf{a}}{\partial t} + (\mathbf{a} \cdot \nabla)\mathbf{a} - \nu \nabla^2 \mathbf{a} = -\frac{1}{\rho}\nabla p + \mathbf{g} \quad in \quad \Omega 1 \tag{4.10b}$$

$$\rho c_p \frac{\partial T}{\partial t} + \rho c_p \mathbf{a} \cdot (\nabla T) - \nabla \cdot (\nabla T) = 0 \quad in \quad \Omega \tag{4.10c}$$

$$T(x, y, z, 0) = 298.15 \quad in \quad \Omega 1 \tag{4.10d}$$

$$T(0, y, z, t) = 298.15 \quad at \quad \partial \Omega 1_{INLET} \tag{4.10e}$$

$$T(x, y, z, t > 5) = 355.15 \quad at \quad \Omega 2 \tag{4.10f}$$

$$U(0, y, z, t < 1) = 16 U_m yz(H-y)(H-z) sin(C\frac{\pi}{2}t)/(H^4), V = 0, Z = 0 \quad at \quad \partial \Omega 1_{INLET} \tag{4.10g}$$

$$U(0, y, z, t \geq 1) = 16 U_m yz(H-y)(H-z)/(H^4), V = 0, Z = 0 \quad at \quad \partial \Omega 1_{INLET} \tag{4.10h}$$

$$V(L, y, z, t) = 0, Z(L, y, z, t) = 0 \quad at \quad \partial \Omega 1_{OUTLET} \tag{4.10i}$$

$$P(L, y, z, t) = Y \rho g_0 \quad at \quad \partial \Omega 1_{OUTLET} \tag{4.10j}$$

where $\Omega 1$ is the body of the heat exchanger domain, $\Omega 2$ is the heated cylinder domain and C is a constant depending on the time step.

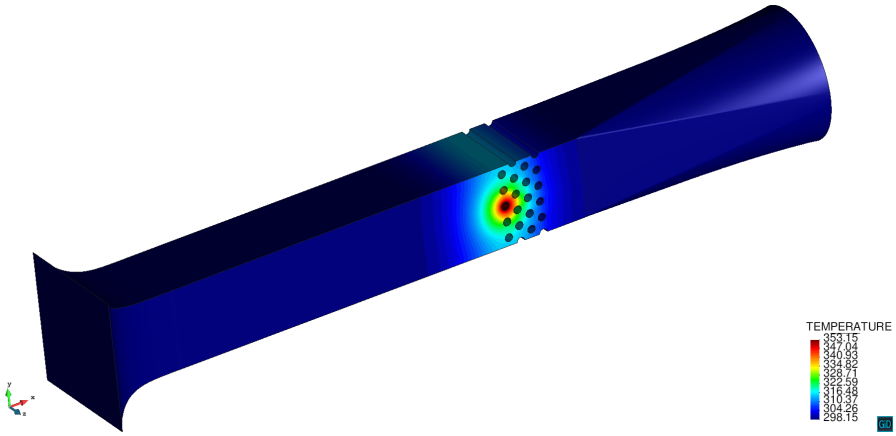
4.6.2 Mesh Size Definition

The reference element size chosen has been $h_{max} = 0.008$. Near the walls and the cylinders the element size has been taken respectively as $h_{min} = 0.001$. Once the mesh has been generated, the cylinder's mesh was composed by 2.842 triangular elements and 16.782 tetrahedral elements and the heat exchanger's mesh by 396.296 triangular elements and 1.944.246 tetrahedral elements. Also for the 3D case, at the interface between the cylinder and the heat exchanger, the two domains have matching nodes.

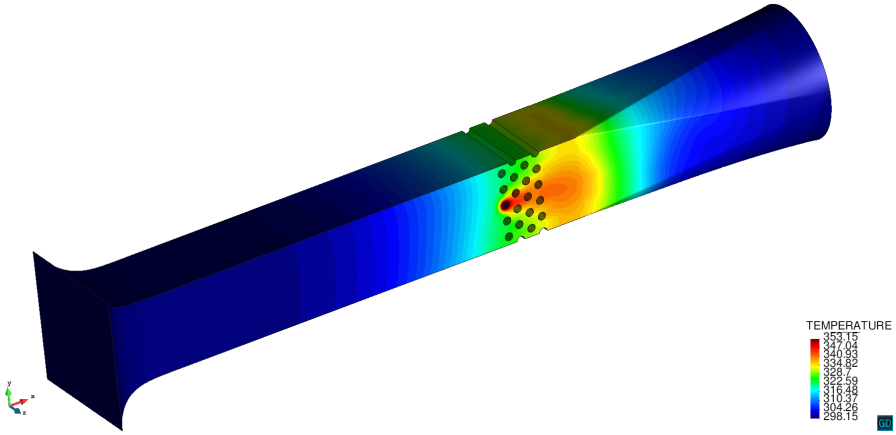
4.6.3 Results

Different snapshots of the temperature evolution of the fluid are collected in Figure 4.28. Energy transfer from the solid to the fluid is also visible for the 3D heat exchanger case.

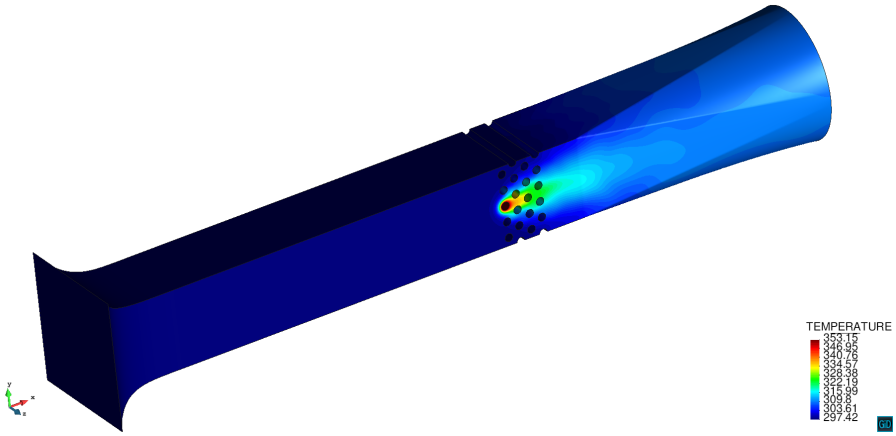
Figure 4.29 shows the the velocity field at different stages. It can be appreciated how vortexes are generated as the velocity increases and also how the velocity increases at the tube banks reaching a maximum velocity of 23.76 km/h.



((a)) t = 0.005 s.

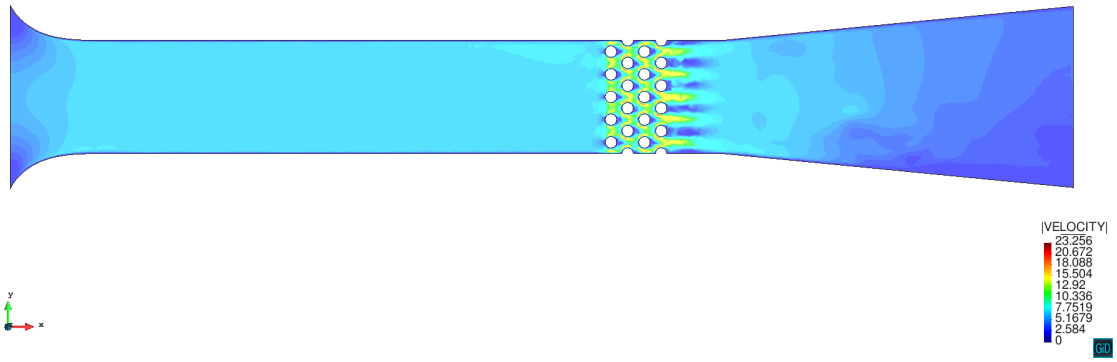


((b)) t = 1 s.

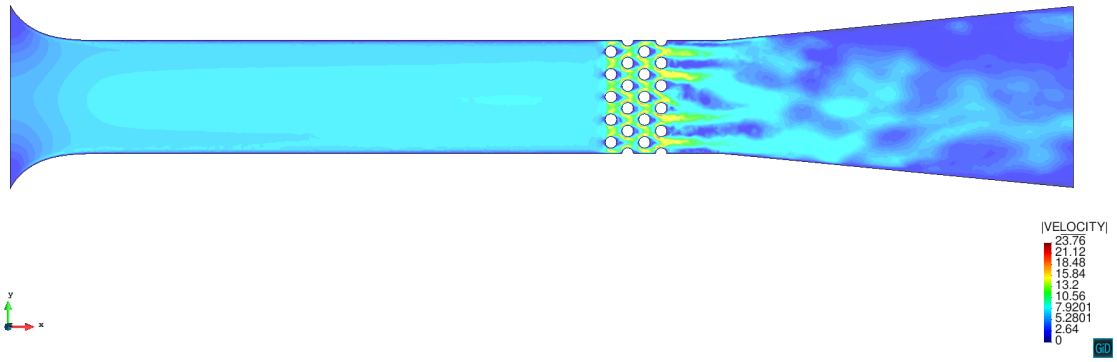


((c)) t = 1.135 s.

Figure 4.28: Temperature evolution of the fluid domain at the heat exchanger in 3D.



((a)) t = 1 s.



((b)) t = 1.35 s.

Figure 4.29: Velocity evolution of the heat exchanger in 3D.

For the solid domain, the heat flux is represented in Figure 4.30. In this case the flux is computed by element, where the value at the nodes is multiplied y the element area.

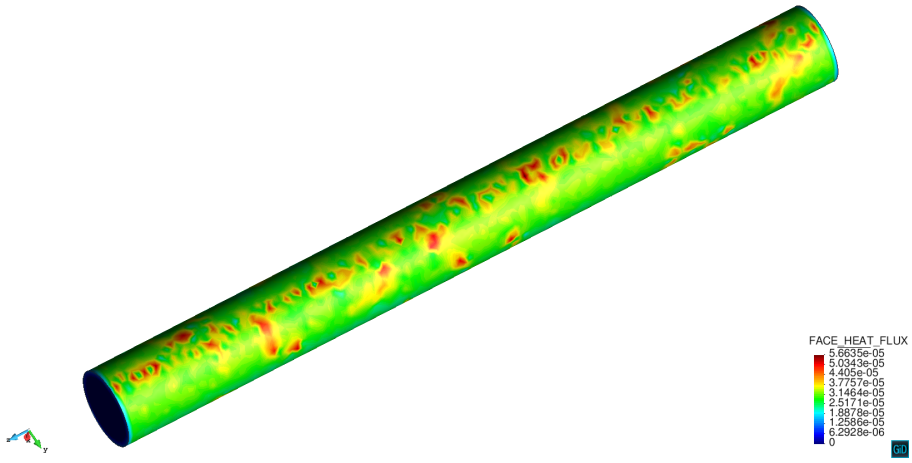


Figure 4.30: Heat flux at t = 1 s.

A temperature decreasing for the solid domain has been registered after 1.35 seconds (Fig. 4.31), where the temperature has decreased by 0.01 K.

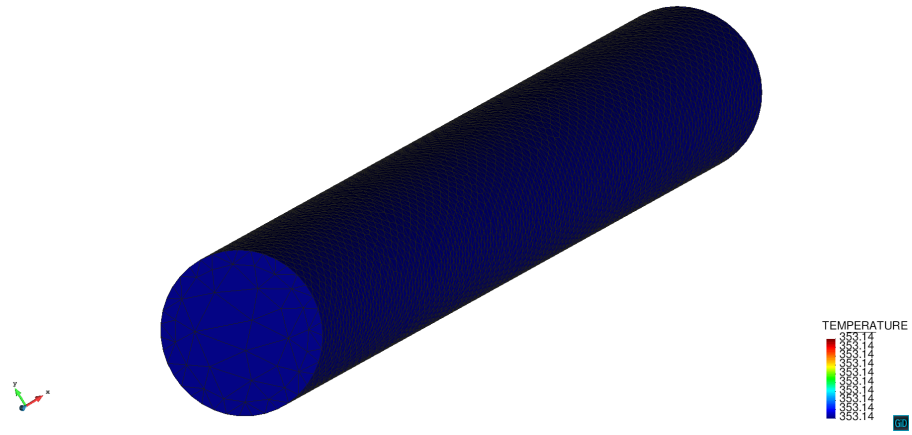


Figure 4.31: Solid temperature field at t = 1.35 s.

Chapter 5

Conclusion

The goal of this thesis has been to develop an application able to study engineering problems where mass and heat transfer are involved. To do that, a solver which couples the fluid dynamics and the heat equation solvers has been implemented in Kratos Multiphysics framework.

First, in order to achieve that, the heat equation application had to be developed in Kratos. Once tested, The heat equation application provided efficient results spanning from the simple pure convection case to a convection dominant case, where the ASGS stabilization technique has been implemented in order to overcome instabilities.

Also the flux condition test, in order to solve problems where Neumann boundary conditions are prescribed, presents results which match perfectly with the analytical given solution.

Heat equation application's reliability put the basis in order to develop the coupled thermal-fluid solver, where the most difficult task has been to couple the two solvers and achieve a converged solution. Results have shown satisfactory shared information between the two solvers, where the fluid domain has experimented exchange of thermal energy by diffusion and convection.

Another achievement has been to simulate problems involving different subdomains with different physics. For this cases heat transfer has been ensured exchanging energy from higher temperature to low temperature at the interface between subdomains. An important strategy in order to reach that has been approaching the problem using the Dirichlet-Neumann method. Here, extremely important concepts as boundary conditions assignation and tolerance criteria in order to achieve reliable solutions have been satisfactory implemented.

Another task has been the simulation of a 2D model without using any turbulent model. The results can be useful in order to compare the different outcome between the VMS formualtion and a turbulent model.

Finally the most important part in order to test the reliability of the thermal-fluid solver has been the numerical study of a 3D real engineering problem. The case of

study has been an heat exchanger working in turbulent regime where air is heated by convection through solid-fluid heat exchange at the interface. Unfortunately, due to the size of the problem and also the high mesh density, the fully copper cooling temperature evolution could not been achieved on time because of the computational limitation and also because the simulation has been run in series.

For the 3D case results show an effective heat transfer by convection for the fluid domain and also a decreasing in temperature have been registered for the solid domain. Unfortunately, due to the high computation required, in order to appreciate

In terms of future lines' work, the first task to be done it is the comparison between the numerical solution and the experimental data in order to see the reliability of the thermal-fluid dynamic solver. Another work to be done in order to improve the efficiency would be the possibility to run the thermal-fluid dynamics solver in MPI parallel environment or bigger machine because simulations become time consuming for large problems.

Finally, the creation of the GiD interface would permits a more friendly approach and less time consuming for the user in order to solve problems where two different domain are involved. Right now the simulation obligates to create two separates problems, implement a Dirichlet-Neumann scheme and also set different functions in order to define the coupling at the interface, which convert the problem tedious and stiff.

Appendix

In this appendix the most important and representative Python and C++ files developed during this work are collected.

A GenerateHeatEquationElement.py

```
//    |  /           |
//    ' /   __| _` | __|  _ \   __|
//    . \   |   (   | |   (   |\__ `
//   _|\_\_|  \__,_|\__|\___/ ____/
//                   Multi-Physics
//
//  License:             BSD License
//  Kratos default license: kratos/license.txt
//
//  Main authors:    Christian Rossi
//

from sympy import *
from KratosMultiphysics import *
from sympy_fe_utilities import *

## Symbolic generation settings
do_simplifications = False
dim_to_compute = "Both"
mode = "c"

if (dim_to_compute == "2D"):
    dim_vector = [2]
elif (dim_to_compute == "3D"):
    dim_vector = [3]
elif (dim_to_compute == "Both"):
    dim_vector = [2,3]

## Read the template file
templatefile = open("heat_equation_template.cpp")
outstring = templatefile.read()
```

```
for dim in dim_vector:
    if(dim == 2):
        nnodes = 3
    elif(dim == 3):
        nnodes = 4

    impose_partion_of_unity = False
    N,DN = DefineShapeFunctions(nnodes, dim, impose_partion_of_unity)

    ## Unknown fields definition
    temp = DefineVector('temp',nnodes)
    tempn = DefineVector('tempn',nnodes)
    tempnn = DefineVector('tempnn',nnodes)

    ## Other simbols definition
    v = DefineMatrix('v',nnodes,dim)
    k = Symbol('k',positive= True)
    cp = Symbol('cp',positive= True)
    rho = Symbol('rho', positive = True)
    dt  = Symbol('dt', positive = True)
    tau = Symbol('tau', positive = True)
    Q = DefineVector('Q',nnodes)

    ## Backward differences coefficients
    bdf0 = Symbol('bdf0')
    bdf1 = Symbol('bdf1')
    bdf2 = Symbol('bdf2')

    ## Test functions definition
    w = DefineVector('w',nnodes)

    ## Data interpolation to the Gauss points
    w_gauss = w.transpose()*N
    Q_gauss = Q.transpose()*N
    v_gauss = v.transpose()*N
    temp_gauss = temp.transpose()*N
    tempder_gauss = (bdf0*temp + bdf1*tempn + bdf2*tempnn)
                    .transpose()*N

    ## Gradients computation (fluid dynamics gradient)
    grad_w = DfiDxj(DN,w) #[1x2]
    grad_temp = DfjDxi(DN,temp) #[2x1]
    div_v = div(DN,v)

    #Terms definition
    w_convective_term = w_gauss*v_gauss.transpose()*grad_temp
    w_diffusive_term = grad_w * k * grad_temp
    w_temporal_term = w_gauss*tempder_gauss
    w_heat_source_term = w_gauss*Q_gauss
```

```
    convective_term = v_gauss.transpose()*grad_temp

    ## Compute galerkin functional
    rv_galerkin =  w_heat_source_term - rho * cp * w_temporal_term...
                  - rho * cp * w_convective_term - w_diffusive_term

    ##  Stabilization functional terms


    Temperature_residual = Q_gauss - rho*cp*(tempder_gauss...
                          + convective_term )

    # Compute the SGS stabilization

    rv_stab +=   tau  *(v_gauss.transpose()* grad_w.transpose() )...
                * Temperature_residual
    ## Add the stabilization terms to the original residual terms
    rv = rv_galerkin  + rv_stab

    ## Define DOFs and test function vectors
    dofs = Matrix( zeros(nnodes, 1))
    testfunc = Matrix( zeros(nnodes, 1))

    for i in range(0,nnodes):

        # TEMPERATURE DOFs and test functions
        dofs[i] = temp[i,0]
        testfunc[i] = w[i,0]

    ## Compute LHS and RHS
    rhs = Compute_RHS(rv.copy(), testfunc, False)
    rhs_out = OutputVector_CollectingFactors(rhs, "rhs", mode)

    # Compute LHS (RHS(residual) differenctiation w.r.t. the DOFs)

    lhs = Compute_LHS(rhs, testfunc, dofs, False)
    lhs_out = OutputMatrix_CollectingFactors(lhs, "lhs", mode)


    if(dim == 2):
        outstring = outstring.replace("//substitute_lhs_2D", lhs_out)
        outstring = outstring.replace("//substitute_rhs_2D", rhs_out)
    elif(dim == 3):
        outstring = outstring.replace("//substitute_lhs_3D", lhs_out)
        outstring = outstring.replace("//substitute_rhs_3D", rhs_out)


## Write the modified template
out = open("heat_equation.cpp",'w')
out.write(outstring)
```

```
out.close()
```

B HeatEquationTemplate.cpp

```
//     /  /              /
//     ' /   __/ _` | __/  _ \   __/
//     . \   /   (   | |   (   |\__ `
//    _/\_\_/   \__,_/\__/\___/ ____/
//                      Multi-Physics
//
//  License:              BSD License
//       Kratos default license: kratos/license.txt
//
//  Main authors:     Christian Rossi
//

#include "custom_elements/heat_equation.h"

namespace Kratos {

  template<>
  void HeatEquation<3>::EquationIdVector(EquationIdVectorType& rResult,
                          ProcessInfo& rCurrentProcessInfo)
  {
      KRATOS_TRY

      unsigned int Dim = 3;
      unsigned int NumNodes = 4;
      unsigned int DofSize  = NumNodes;

      if (rResult.size() != DofSize)
          rResult.resize(DofSize, false);

      for(unsigned int i=0; i<NumNodes; i++)
      {

          rResult[i]  =  this->GetGeometry()[i].GetDof(TEMPERATURE).
                                  EquationId();
      }

      KRATOS_CATCH("")
  }


template<>
void HeatEquation<2>::EquationIdVector(EquationIdVectorType& rResult,
                          ProcessInfo& rCurrentProcessInfo)
{
    KRATOS_TRY
```

```
    unsigned int Dim = 2;
    unsigned int NumNodes = 3;
    unsigned int DofSize  = NumNodes;

    if (rResult.size() != DofSize)
        rResult.resize(DofSize, false);

    for(unsigned int i=0; i<NumNodes; i++)
    {
        rResult[i]  =  this->GetGeometry()[i].
                       GetDof(TEMPERATURE).EquationId();

    }

    KRATOS_CATCH("")
}


template<>
void HeatEquation<3>::GetDofList(DofsVectorType& ElementalDofList,
                       ProcessInfo& rCurrentProcessInfo)
{
    KRATOS_TRY

    unsigned int Dim = 3;
    unsigned int NumNodes = 4;
    unsigned int DofSize  = NumNodes;

    if (ElementalDofList.size() != DofSize)
        ElementalDofList.resize(DofSize);

    for(unsigned int i=0; i<NumNodes; i++)
    {
        ElementalDofList[i ]  =  this->GetGeometry()[i].
                                 pGetDof(TEMPERATURE);

    }

    KRATOS_CATCH("");
}

template<>
void HeatEquation<2>::GetDofList(DofsVectorType& ElementalDofList,
                       ProcessInfo& rCurrentProcessInfo)
{
    KRATOS_TRY

    unsigned int Dim = 2;
    unsigned int NumNodes = 3;
    unsigned int DofSize  = NumNodes;
```

```
    if (ElementalDofList.size() != DofSize)
        ElementalDofList.resize(DofSize);

    for(unsigned int i=0; i<NumNodes; i++)
    {

        ElementalDofList[i] = this->GetGeometry()[i].
                              pGetDof(TEMPERATURE);
    }

    KRATOS_CATCH("");
}



template<>
void HeatEquation<3>::ComputeGaussPointLHSContribution
    (bounded_matrix<double,4,4>& lhs, const ElementDataStruct& data)
{
    const int nnodes = 4;
    const int dim = 3;
    const double rho = data.rho;
    const double k = data.k;
    const double cp = data.cp;
    const double& bdf0 = data.bdf0;
    const double& bdf1 = data.bdf1;
    const double& bdf2 = data.bdf2;
    const double h = data.h;
    const double& delta_t = data.delta_t;
    const array_1d<double,nnodes>& Q = data.Q;
    const bounded_matrix<double,nnodes,dim>& v = data.v;
    const array_1d<double,nnodes>& temp = data.temp;
    const array_1d<double,nnodes>& tempn = data.tempn;
    const array_1d<double,nnodes>& tempnn = data.tempnn;
    const double& dyn_tau_coeff = data.dyn_tau_coeff;

    // Get shape function values
    const array_1d<double,nnodes>& N = data.N;
    const bounded_matrix<double,nnodes,dim>& DN = data.DN_DX;

    const array_1d<double,dim> v_gauss = prod(trans(v), N);
    const double v_norm = norm_2(v_gauss);
    // Stabilization parameters
    const double c1 = 2.0;
    const double c2 = 4.0;
    const double tau = 1.0/(dyn_tau_coeff*rho*cp/delta_t +
                        c1*rho*cp*v_norm/h + c2*k/(h*h));
```

```
}


template<>
void HeatEquation<2>::ComputeGaussPointLHSContribution
    (bounded_matrix<double,3,3>& lhs, const ElementDataStruct& data)
{
    const int nnodes = 3;
    const int dim = 2;
    const double rho = data.rho;
    const double k = data.k;
    const double cp = data.cp;
    const double h = data.h;
    const double& delta_t = data.delta_t;
    const double& bdf0 = data.bdf0;
    const double& bdf1 = data.bdf1;
    const double& bdf2 = data.bdf2;

    const array_1d<double,nnodes>& Q = data.Q;
    const bounded_matrix<double,nnodes,dim>& v = data.v;
    const array_1d<double,nnodes>& temp = data.temp;
    const array_1d<double,nnodes>& tempn = data.tempn;
    const array_1d<double,nnodes>& tempnn = data.tempnn;
    const double& dyn_tau_coeff = data.dyn_tau_coeff;



    // Get shape function values
    const array_1d<double,nnodes>& N = data.N;
    const bounded_matrix<double,nnodes,dim>& DN = data.DN_DX;

    // Stabilization parameters
    const array_1d<double,dim> v_gauss = prod(trans(v), N);
    const double v_norm = norm_2(v_gauss);
    // Stabilization parameters
    const double c1 = 2.0;
    const double c2 = 4.0;
    const double tau = 1.0/(dyn_tau_coeff*rho*cp/delta_t
                      + c1*rho*cp*v_norm/h + c2*k/(h*h));


}


template<>
void HeatEquation<3>::ComputeGaussPointRHSContribution
    (array_1d<double,4>& rhs, const ElementDataStruct& data)
{
    const int nnodes = 4;
```

```
    const int dim = 3;
    const double rho = data.rho;
    const double k = data.k;
    const double cp = data.cp;
    const double h = data.h;
    const double& delta_t = data.delta_t;
    const double& bdf0 = data.bdf0;
    const double& bdf1 = data.bdf1;
    const double& bdf2 = data.bdf2;
    const array_1d<double,nnodes>& Q = data.Q;
    const bounded_matrix<double,nnodes,dim>& v = data.v;
    const array_1d<double,nnodes>& temp = data.temp;
    const array_1d<double,nnodes>& tempn = data.tempn;
    const array_1d<double,nnodes>& tempnn = data.tempnn;
    const double& dyn_tau_coeff = data.dyn_tau_coeff;

    // Get shape function values
    const array_1d<double,nnodes>& N = data.N;
    const bounded_matrix<double,nnodes,dim>& DN = data.DN_DX;

    // Auxiliary variables used in the calculation of the RHS
    const double q_gauss = inner_prod(data.Q, data.N);


    // Stabilization parameters
    const array_1d<double,dim> v_gauss = prod(trans(v), N);
    const double v_norm = norm_2(v_gauss);
    // Stabilization parameters
    const double c1 = 2.0;
    const double c2 = 4.0;
    const double tau = 1.0/(dyn_tau_coeff*rho*cp/delta_t +
                      c1*rho*cp*v_norm/h + c2*k/(h*h));

}


template<>
void HeatEquation<2>::ComputeGaussPointRHSContribution
    (array_1d<double,3>& rhs, const ElementDataStruct& data)
{
    const int nnodes = 3;
    const int dim = 2;
    const double rho = data.rho;
    const double k = data.k;
    const double cp = data.cp;
    const double h = data.h;
    const double& delta_t = data.delta_t;
    const double& bdf0 = data.bdf0;
    const double& bdf1 = data.bdf1;
    const double& bdf2 = data.bdf2;
```

```
    const array_1d<double,nnodes>& Q = data.Q;
    const bounded_matrix<double,nnodes,dim>& v = data.v;
    const array_1d<double,nnodes>& temp = data.temp;
    const array_1d<double,nnodes>& tempn = data.tempn;
    const array_1d<double,nnodes>& tempnn = data.tempnn;
    const double& dyn_tau_coeff = data.dyn_tau_coeff;

    // Get shape function values
    const array_1d<double,nnodes>& N = data.N;
    const bounded_matrix<double,nnodes,dim>& DN = data.DN_DX;

    // Auxiliary variables used in the calculation of the RHS

    const double q_gauss = inner_prod(data.Q, data.N);


    // Stabilization parameters
    const array_1d<double,dim> v_gauss = prod(trans(v), N);
    const double v_norm = norm_2(v_gauss);
    // Stabilization parameters
    const double c1 = 2.0;
    const double c2 = 4.0;
    const double tau = 1.0/(dyn_tau_coeff*rho*cp/delta_t
                      + c1*rho*cp*v_norm/h + c2*k/(h*h));

}

}
```

C HeatEquationSolver.py

```
from __future__ import print_function, absolute_import, division
# importing the Kratos Library
import KratosMultiphysics
import KratosMultiphysics.HeatEquationApplication
#import KratosMultiphysics.MeshingApplication as KratosMeshing

# Check that KratosMultiphysics was imported in the main script
KratosMultiphysics.CheckForPreviousImport()

def CreateSolver(main_model_part, custom_settings):
    return HeatEquationSolver(main_model_part, custom_settings)

class HeatEquationSolver(object):

    def __init__(self, main_model_part, custom_settings):

         object is implemented
        self.main_model_part = main_model_part
```

```
        ##settings string in json format
        base_settings = KratosMultiphysics.Parameters("""
␣␣␣␣␣␣␣␣{
␣␣␣␣␣␣␣␣␣␣␣␣"solver_type"␣:␣"heat_equation_solver",
␣␣␣␣␣␣␣␣␣␣␣␣"problem_data"␣:{
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣"domain_size"␣:␣2
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣},

␣␣␣␣␣␣␣␣␣␣␣␣"model_import_settings":␣{
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣"input_type":␣"mdpa",
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣"input_filename":␣"unknown_name"
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣},
␣␣␣␣␣␣␣␣␣␣␣␣"relative_tolerance"␣␣␣␣␣␣␣␣:␣1e-5,
␣␣␣␣␣␣␣␣␣␣␣␣"absolute_tolerance"␣␣␣␣␣␣␣␣:␣1e-7,
␣␣␣␣␣␣␣␣␣␣␣␣"time_order"␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣:␣2,
␣␣␣␣␣␣␣␣␣␣␣␣"echo_level"␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣:␣0,
␣␣␣␣␣␣␣␣␣␣␣␣"maximum_iterations"␣␣␣␣␣␣␣␣:␣20,
␣␣␣␣␣␣␣␣␣␣␣␣"compute_reactions"␣␣␣␣␣␣␣␣␣␣:␣false,
␣␣␣␣␣␣␣␣␣␣␣␣"reform_dofs_at_each_step"␣␣:␣false,
␣␣␣␣␣␣␣␣␣␣␣␣"move_mesh_flag"␣␣␣␣␣␣␣␣␣␣␣␣:␣false,
␣␣␣␣␣␣␣␣␣␣␣␣"domain_model_part"␣␣␣␣␣␣␣␣␣:␣"your_computational_domain",
␣␣␣␣␣␣␣␣␣␣␣␣"dynamic_tau"␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣:␣1
␣␣␣␣␣␣␣␣}""")

        ## Overwrite the default settings with user-provided parameters
        self.settings = custom_settings
        self.settings.ValidateAndAssignDefaults(base_settings)

        ## Construct the linear solver

        #import linear_solver_factory
        #self.linear_solver = linear_solver_factory.
        ConstructSolver(self.settings
        ["linear_solver_settings"])
        from KratosMultiphysics.ExternalSolversApplication
        import SuperLUIterativeSolver
        self.linear_solver = SuperLUIterativeSolver()

        print("Construction␣of␣HeatEquationSolver␣finished")



    def AddVariables(self):
        ## Add base class variables
        self.main_model_part.AddNodalSolutionStepVariable
        (KratosMultiphysics.DENSITY);
        self.main_model_part.AddNodalSolutionStepVariable
        (KratosMultiphysics.CONDUCTIVITY);
        self.main_model_part.AddNodalSolutionStepVariable
        (KratosMultiphysics.SPECIFIC_HEAT);
```

```
        self.main_model_part.AddNodalSolutionStepVariable
        (KratosMultiphysics.HEAT_FLUX);
        self.main_model_part.AddNodalSolutionStepVariable
        (KratosMultiphysics.TEMPERATURE);
        self.main_model_part.AddNodalSolutionStepVariable
        (KratosMultiphysics.VELOCITY);
        self.main_model_part.AddNodalSolutionStepVariable
        (KratosMultiphysics.NODAL_AREA)
        self.main_model_part.AddNodalSolutionStepVariable
        (KratosMultiphysics.FACE_HEAT_FLUX )
        print("Base class Heat Equation solver variable added correctly")

    def ImportModelPart(self):
        ## Read model part
        self._ModelPartReading()
        ## Replace elements and conditions
        self._ExecuteAfterReading()
        ## Set buffer size
        self._SetBufferSize()

        # print(self.main_model_part.GetSubModelPart("Wall_solid"))
        print ("Base class model reading finished.")

    def ExportModelPart(self):
        name_out_file = self.settings["model_import_settings"]
        ["input_filename"].GetString()+".out"
        file = open(name_out_file + ".mdpa","w")
        file.close()

        ## Model part writing
        KratosMultiphysics.ModelPartIO(name_out_file,

        KratosMultiphysics.IO.WRITE).WriteModelPart(self.main_model_part)

    def AddDofs(self):
        ## Adding dofs
        for node in self.main_model_part.Nodes:
            node.AddDof(KratosMultiphysics.TEMPERATURE,

            KratosMultiphysics.FACE_HEAT_FLUX)

        print("Base class Heat Equation solver DOFs added correctly.")

    def AdaptMesh(self):
        pass

    def GetMinimumBufferSize(self):
        return 3

    def GetComputingModelPart(self):
```

```
        return self.main_model_part #.GetSubModelPart(self.settings
        ["domain_model_part"].GetString())

    def GetOutputVariables(self):
        pass

    def Initialize(self):
        self.computing_model_part = self.GetComputingModelPart()

        # Creating the solution strategy
        self.conv_criteria = KratosMultiphysics.ResidualCriteria
        (self.settings["relative_tolerance"].GetDouble(),
self.settings["absolute_tolerance"].GetDouble())

        self.bdf_process = KratosMultiphysics.
        ComputeBDFCoefficientsProcess(self.computing_model_part,
        self.settings["time_order"].GetInt())

        time_scheme = KratosMultiphysics.
        ResidualBasedIncrementalUpdateStaticScheme()

        builder_and_solver = KratosMultiphysics.
        ResidualBasedBlockBuilderAndSolver(self.linear_solver)

        self.solver=KratosMultiphysics.
        ResidualBasedNewtonRaphsonStrategy(self.computing_model_part,
        time_scheme,
        self.linear_solver,
        self.conv_criteria,
        builder_and_solver,
        self.settings["maximum_iterations"].GetInt(),
        self.settings["compute_reactions"].GetBool()
self.settings["reform_dofs_at_each_step"].GetBool(),
        self.settings["move_mesh_flag"].GetBool())

        (self.solver).SetEchoLevel(self.settings["echo_level"].GetInt())



        KratosMultiphysics.CalculateNodalAreaProcess
        (self.computing_model_part,
         self.settings["problem_data"]["domain_size"].GetInt()).Execute()

        (self.solver).Initialize()
        (self.solver).Check()


    def SaveRestart(self):
        pass #one should write the restart file here
```

```
    def Clear(self):
        (self.solver).Clear()

    def Check(self):
        (self.solver).Check()

    def SetEchoLevel(self, level):
        (self.solver).SetEchoLevel(level)

    def SolverInitialize(self):
        (self.solver).Initialize()

    def SolverInitializeSolutionStep(self):
        (self.bdf_process).Execute()
        (self.solver).InitializeSolutionStep()

    def SolverPredict(self):
        (self.solver).Predict()

    def SolverSolveSolutionStep(self):
        (self.solver).SolveSolutionStep()

    def SolverFinalizeSolutionStep(self):
        (self.solver).FinalizeSolutionStep()

    def Solve(self):

        self.SolverInitializeSolutionStep()

        self.SolverPredict()

        self.SolverSolveSolutionStep()

        self.SolverFinalizeSolutionStep()

    def _ModelPartReading(self):
        ## Model part reading
        if(self.settings["model_import_settings"]["input_type"]
        .GetString() == "mdpa"):
            KratosMultiphysics.ModelPartIO
            (self.settings["model_import_settings"]
            ["input_filename"].GetString()).
            ReadModelPart(self.main_model_part)
        else:
            raise Exception
            ("Other input options are not yet implemented.")

    def _ExecuteAfterReading(self):

        for el in self.main_model_part.Elements:
```

```
            rho = el.Properties.GetValue(KratosMultiphysics.DENSITY)
            k = el.Properties.GetValue(KratosMultiphysics.CONDUCTIVITY)
            cp = el.Properties.GetValue(KratosMultiphysics.SPECIFIC_HEAT)
            break

        print(self.settings["dynamic_tau"].GetDouble())
        self.GetComputingModelPart().ProcessInfo[KratosMultiphysics.TAU]
        = self.settings["dynamic_tau"].GetDouble()

        KratosMultiphysics.VariableUtils()
        .SetScalarVar(KratosMultiphysics.DENSITY, rho,

        self.main_model_part.Nodes)
        KratosMultiphysics.VariableUtils()
        .SetScalarVar(KratosMultiphysics.CONDUCTIVITY, k,

        self.main_model_part.Nodes)
        KratosMultiphysics.VariableUtils()
        .SetScalarVar(KratosMultiphysics.SPECIFIC_HEAT, cp,

        self.main_model_part.Nodes)

    def _SetBufferSize(self):
        ## Set the buffer size
        current_buffer_size = self.main_model_part.GetBufferSize()
        if(self.GetMinimumBufferSize() > current_buffer_size):
            self.main_model_part.SetBufferSize
            (self.GetMinimumBufferSize())
```

D CFD-HeatTransferSolver.py

```
# importing the Kratos Library
import KratosMultiphysics
import KratosMultiphysics.FluidDynamicsApplication as KratosCFD
import KratosMultiphysics.HeatEquationApplication as KratosHEq
KratosMultiphysics.CheckForPreviousImport()

import navier_stokes_base_solver
def CreateSolver(fluid_main_model_part, heat_equation_main_model_part,

project_parameters):
        return CFDThermalSolver(fluid_main_model_part,

        heat_equation_main_model_part, project_parameters)

class CFDThermalSolver:
    """ˍˍˍˍˍˍˍˍˍˍ"""
    def __init__(self, fluid_main_model_part,
```

```
    heat_equation_main_model_part, project_parameters ):

        # Initial tests

        self.fluid_main_model_part = fluid_main_model_part
        self.heat_equation_main_model_part =
             heat_equation_main_model_part

        solvers_default_settings = KratosMultiphysics.Parameters("""
␣␣␣␣␣␣␣␣{
␣␣␣␣␣␣␣␣"heat_equation_solver_settings":
␣␣␣␣␣␣␣␣␣␣␣␣{
␣␣␣␣␣␣␣␣␣␣␣␣"solver_type":␣"heat_equation_solver",
␣␣␣␣␣␣␣␣␣␣␣␣"problem_data"␣:{
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣"domain_size"␣:␣3
␣␣␣␣␣␣␣␣␣␣␣␣},
␣␣␣␣␣␣␣␣␣␣␣␣"model_import_settings":␣{
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣"input_type":␣"mdpa",
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣"input_filename":␣"coupled_solid"
␣␣␣␣␣␣␣␣␣␣␣␣},
␣␣␣␣␣␣␣␣␣␣␣␣"relative_tolerance"␣␣␣␣␣␣␣␣:␣1e-5,
␣␣␣␣␣␣␣␣␣␣␣␣"absolute_tolerance"␣␣␣␣␣␣␣␣:␣1e-7,
␣␣␣␣␣␣␣␣␣␣␣␣"time_order"␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣:␣2,
␣␣␣␣␣␣␣␣␣␣␣␣"echo_level"␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣:␣0,
␣␣␣␣␣␣␣␣␣␣␣␣"maximum_iterations"␣␣␣␣␣␣␣␣:␣20,
␣␣␣␣␣␣␣␣␣␣␣␣"compute_reactions"␣␣␣␣␣␣␣␣␣␣:␣true,
␣␣␣␣␣␣␣␣␣␣␣␣"reform_dofs_at_each_step"␣␣:␣false,
␣␣␣␣␣␣␣␣␣␣␣␣"move_mesh_flag"␣␣␣␣␣␣␣␣␣␣␣␣:␣false,
␣␣␣␣␣␣␣␣␣␣␣␣"domain_model_part"␣␣␣␣␣␣␣␣␣␣:␣"your_computational_domain",
␣␣␣␣␣␣␣␣␣␣␣␣"dynamic_tau"␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣:␣1
␣␣␣␣␣␣␣␣␣␣␣␣},
␣␣␣␣␣␣␣␣"fluid_solver_settings":
␣␣␣␣␣␣␣␣␣␣␣␣{
␣␣␣␣␣␣␣␣␣␣␣␣"solver_type":␣"navier_stokes_solver_vmsmonolithic",
␣␣␣␣␣␣␣␣␣␣␣␣"model_import_settings":{
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣"input_type":␣"mdpa",
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣"input_filename":␣"coupled_fluid_vel"
␣␣␣␣␣␣␣␣␣␣␣␣},
␣␣␣␣␣␣␣␣␣␣␣␣"maximum_iterations":␣10,
␣␣␣␣␣␣␣␣␣␣␣␣"dynamic_tau":␣0.0,
␣␣␣␣␣␣␣␣␣␣␣␣"oss_switch":␣0,
␣␣␣␣␣␣␣␣␣␣␣␣"echo_level":␣0,
␣␣␣␣␣␣␣␣␣␣␣␣"consider_periodic_conditions":␣false,
␣␣␣␣␣␣␣␣␣␣␣␣"compute_reactions":␣true,
␣␣␣␣␣␣␣␣␣␣␣␣"divergence_clearance_steps":␣0,
␣␣␣␣␣␣␣␣␣␣␣␣"reform_dofs_at_each_step":␣true,
␣␣␣␣␣␣␣␣␣␣␣␣"relative_velocity_tolerance":␣1e-3,
␣␣␣␣␣␣␣␣␣␣␣␣"absolute_velocity_tolerance":␣1e-5,
␣␣␣␣␣␣␣␣␣␣␣␣"relative_pressure_tolerance":␣1e-3,
␣␣␣␣␣␣␣␣␣␣␣␣"absolute_pressure_tolerance":␣1e-5,
```

```
            "linear_solver_settings"        :
            {
                "solver_type" : "AMGCL",
                "smoother_type":"ilu0",
                "krylov_type": "gmres",
                "coarsening_type": "aggregation",
                "max_iteration": 200,
                "provide_coordinates": false,
                "gmres_krylov_space_dimension": 100,
                "verbosity" : 0,
                "tolerance": 1e-7,
                "scaling": false,
                "block_size": 1,
                "use_block_matrices_if_possible" : true,
                "coarse_enough" : 5000
            },
            "volume_model_part_name" : "volume_model_part",
            "skin_parts": [""],
            "no_skin_parts":[""],
            "time_stepping"                 :
            {
                "automatic_time_step" : false,
                "CFL_number"          : 1,
                "minimum_delta_time"  : 1e-4,
                "maximum_delta_time"  : 0.01,
                "user_delta_time"     : 0.004
            },
            "alpha":-0.3,
            "move_mesh_strategy": 0,
            "periodic": "periodic",
            "move_mesh_flag": false,
            "turbulence_model": "None"
        }
    }""")



        self.settings = KratosMultiphysics.Parameters("{}")
        self.settings.AddValue("heat_equation_solver_settings",
        project_parameters["heat_settings"]
        ["heat_equation_solver_settings"])
        self.settings.AddValue("fluid_solver_settings",
        project_parameters["fluid_settings"]["fluid_solver_settings"])

        # Overwrite the default settings with user-provided parameters
        self.settings.RecursivelyValidateAndAssignDefaults
        (solvers_default_settings)
        # Auxiliar variables

        print("*** Partitioned CFDThermalSolver  construction starts...")
```

```
        # Construct the fluid solver
        fluid_solver_module =

        __import__(self.settings["fluid_solver_settings"]

        ["solver_type"].GetString())
        self.fluid_solver=fluid_solver_module.
        CreateSolver(self.fluid_main_model_part,
        self.settings["fluid_solver_settings"])

        print("* Fluid solver constructed.")

        # Construct the heat_equation solver
        heat_equation_solver_module = __import__(self.settings
        ["heat_equation_solver_settings"]
        ["solver_type"].GetString())
        self.heat_equation_solver = heat_equation_solver_module.
        CreateSolver(self.heat_equation_main_model_part,

        self.settings["heat_equation_solver_settings"])

        print("* Heat solver constructed.")


    def GetMinimumBufferSize(self):
        # Get fluid buffer size
        buffer_fluid = self.fluid_solver.GetMinimumBufferSize()
        # Get heat equation buffer size
        buffer_heat = self.heat_equation_solver.GetMinimumBufferSize()

        return min(buffer_heat,buffer_fluid)

    def AddVariables(self):

        self.fluid_solver.AddVariables()
        self.fluid_solver.main_model_part.AddNodalSolutionStepVariable
        (KratosMultiphysics.TEMPERATURE)
        self.fluid_solver.main_model_part.AddNodalSolutionStepVariable
        (KratosMultiphysics.CONDUCTIVITY)
        self.fluid_solver.main_model_part.AddNodalSolutionStepVariable
        (KratosMultiphysics.SPECIFIC_HEAT)
        self.fluid_solver.main_model_part.AddNodalSolutionStepVariable
        (KratosMultiphysics.HEAT_FLUX)
        self.heat_equation_solver.AddVariables()


    def ImportModelPart(self):
        #read fluid
        self.fluid_solver.ImportModelPart()
        # print(self.fluid_main_model_part)
```

```
        GetSubModelPart("Wall_solid"))

        mpfluid = self.fluid_solver.main_model_part

        mpheat = self.heat_equation_solver.main_model_part


        Dim =self.settings["heat_equation_solver_settings"]
        ["problem_data"]["domain_size"].GetInt()
        modeler = KratosMultiphysics.ConnectivityPreserveModeler()
        if (Dim == 2):
            modeler.GenerateModelPart(mpfluid,
                                      mpheat,
                                      "HeatEquation2D",
                                      "LineCondition2D2N")
        else:
            modeler.GenerateModelPart(mpfluid,
                                      mpheat,
                                     "HeatEquation3D",
                                     "WallCondition3D3N")



        #print(self.heat_equation_solver.main_model_part.
        GetSubModelPart("Wall_solid"))

        print("********ConnectivityPreserveModeler done!***********")

        KratosMultiphysics.VariableUtils().SetScalarVar(KratosMultiphysics

        .DENSITY, 1.225, self.heat_equation_solver.main_model_part.Nodes)
        KratosMultiphysics.VariableUtils().SetScalarVar(KratosMultiphysics

        .CONDUCTIVITY, 0.0257, self.heat_equation_solver.main_model_part.Nodes)
        KratosMultiphysics.VariableUtils().SetScalarVar(KratosMultiphysics

        .SPECIFIC_HEAT, 1.005, self.heat_equation_solver.main_model_part.Nodes)
        KratosMultiphysics.VariableUtils().SetScalarVar(KratosMultiphysics

        .HEAT_FLUX, 0, self.heat_equation_solver.main_model_part.Nodes)



    def AddDofs(self):

        # Add DOFs fluid
        self.fluid_solver.AddDofs()
        # Add DOFs heat_equation
```

```
        self.heat_equation_solver.AddDofs()


    def Initialize(self):
        # Initialize fluid solver
        self.fluid_solver.Initialize()
        # Initialize heat_equation solver
        self.heat_equation_solver.Initialize()

    def SolverInitializeSolutionStep(self):
        # (self.bdf_process).Execute()
        self.fluid_solver.InitializeSolutionStep()
        self.heat_equation_solver.SolverInitializeSolutionStep()
    #
    def SolverPredict(self):
        self.fluid_solver.Predict()
        self.heat_equation_solver.SolverPredict()

    def SolverSolveSolutionStep(self):
        self.fluid_solver.SolveSolutionStep()
        self.heat_equation_solver.SolverSolveSolutionStep()

    def SolverFinalizeSolutionStep(self):
        self.fluid_solver.FinalizeSolutionStep()
        self.heat_equation_solver.SolverFinalizeSolutionStep()

    def GetComputingModelPart(self):
        return self.fluid_solver.GetComputingModelPart()


    def GetOutputVariables(self):
        pass



    def SaveRestart(self):
        pass


    def Solve(self):
        print("-------------------fluid solving")
        self.fluid_solver.Solve()
        print("-------------------Thermal solving")
        self.heat_equation_solver.Solve()
```

E HeatEquationNeumannCondition.cpp

```
//     |  /           |
//     ' /   __| _` | __|  _ \   __|
//     . \   |   (   | |   (   |\__ \.
```

```
//   _/\_\_/  \__,_/\__/\___/ ____/
//                   Multi-Physics
//
//  License:              BSD License
//       Kratos default license: kratos/license.txt
//
//  Main authors:    Christian Rossi
//

#include "heat_equation_Neumann_condition.h"

namespace Kratos
{


template <>
void HeatEquationNeumannCondition<2,2>::EquationIdVector
     (EquationIdVectorType& rResult,
     ProcessInfo& rCurrentProcessInfo)
{
    const unsigned int NumNodes = 2;
    const unsigned int LocalSize = 2;
    unsigned int LocalIndex = 0;

    if (rResult.size() != LocalSize)
        rResult.resize(LocalSize, false);

    for (unsigned int iNode = 0; iNode < NumNodes; ++iNode)
    {
        rResult[LocalIndex++] = this->GetGeometry()[iNode].
        GetDof(TEMPERATURE).EquationId();


    }
}

/**
 * @see HeatEquationNeumannCondition::EquationIdVector
 */
template <>
void HeatEquationNeumannCondition<3,3>::EquationIdVector
     (EquationIdVectorType& rResult,
      ProcessInfo& rCurrentProcessInfo)
{
    const SizeType NumNodes = 3;
    const SizeType LocalSize = 3;
    unsigned int LocalIndex = 0;

    if (rResult.size() != LocalSize)
        rResult.resize(LocalSize, false);
```

```
    for (unsigned int iNode = 0; iNode < NumNodes; ++iNode)
    {

        rResult[LocalIndex++] = this->GetGeometry()[iNode].
        GetDof(TEMPERATURE).EquationId();
    }
}

/**
 * @see HeatEquationNeumannCondition::GetDofList
 */
template <>
void HeatEquationNeumannCondition<2,2>::GetDofList
     (DofsVectorType& rElementalDofList,
      ProcessInfo& rCurrentProcessInfo)
{
    const SizeType NumNodes = 2;
    const SizeType LocalSize = 2;

    if (rElementalDofList.size() != LocalSize)
        rElementalDofList.resize(LocalSize);

    unsigned int LocalIndex = 0;

    for (unsigned int iNode = 0; iNode < NumNodes; ++iNode)
    {

        rElementalDofList[LocalIndex++] = this->GetGeometry()
        [iNode].pGetDof(TEMPERATURE);
    }
}

/**
 * @see HeatEquationNeumannCondition::GetDofList
 */
template <>
void HeatEquationNeumannCondition<3,3>::GetDofList
     (DofsVectorType& rElementalDofList,
      ProcessInfo& rCurrentProcessInfo)
{
    const SizeType NumNodes = 3;
    const SizeType LocalSize = 3;

    if (rElementalDofList.size() != LocalSize)
        rElementalDofList.resize(LocalSize);

    unsigned int LocalIndex = 0;

    for (unsigned int iNode = 0; iNode < NumNodes; ++iNode)
```

```
    {

        rElementalDofList[LocalIndex++] = this->GetGeometry()[iNode].
        pGetDof(TEMPERATURE);
    }
}

/// Computes the Gauss pt. LHS contribution
/**
* @param lhs_gauss reference to the local LHS matrix
* @param data Gauss pt. data structure
*/
template<unsigned int Tdim, unsigned int TNumNodes>
void HeatEquationNeumannCondition<Tdim,TNumNodes>::7
     ComputeGaussPointLHSContribution
     (bounded_matrix<double,TNumNodes,TNumNodes>& lhs_gauss,
const ConditionDataStruct& data)
{
    noalias(lhs_gauss) = ZeroMatrix(TNumNodes,TNumNodes);
}/// Computes the Gauss pt. RHS contribution
/**
* @param rhs_gauss reference to the local RHS vector
* @param data Gauss pt. data structure
*/
template<unsigned int TDim, unsigned int TNumNodes>
void HeatEquationNeumannCondition<TDim,TNumNodes>::
     ComputeGaussPointRHSContribution
     (array_1d<double,TNumNodes>& rhs_gauss,
     const ConditionDataStruct& data)
{

    // Initialize the local RHS
    noalias(rhs_gauss) = ZeroVector(TNumNodes);

    // Gauss pt. Neumann BC contribution
    this->ComputeRHSNeumannContribution(rhs_gauss, data);
}

/// Computes the condition RHS Neumann BC contribution
/**
* @param rhs_gauss reference to the local RHS vector
* @param data Gauss pt. data structure
*/
template<unsigned int Tdim, unsigned int TNumNodes>
void HeatEquationNeumannCondition<Tdim,TNumNodes>::
     ComputeRHSNeumannContribution(array_1d<double,
     TNumNodes>& rhs_gauss,
     const ConditionDataStruct& data)
{
    const GeometryType& rGeom = this->GetGeometry();
```

```
    // Add Neumann BC contribution
    for (unsigned int i=0; i<TNumNodes; ++i)
    {
        const double flux_ext = rGeom[i].
        FastGetSolutionStepValue(FACE_HEAT_FLUX);
        const double k = 401; //rGeom[i].
        FastGetSolutionStepValue(CONDUCTIVITY);
        rhs_gauss[i] -= k*data.wGauss*data.N[i]*flux_ext;
        //std::cout << "Neumann contribution OK\n";
    }
}

/// Computes the 2D condition normal
/**
* @param An reference to condition normal vector
*/
template <>
void HeatEquationNeumannCondition<2,2>::
CalculateNormal(array_1d<double,3>& An)
{
    Geometry<Node<3> >& pGeometry = this->GetGeometry();

    An[0] =    pGeometry[1].Y() - pGeometry[0].Y();
    An[1] = - (pGeometry[1].X() - pGeometry[0].X());
    An[2] =     0.00;

}

/// Computes the 3D condition normal
/**
* @param An reference to condition normal vector
*/
template <>
void HeatEquationNeumannCondition<3,3>::
CalculateNormal(array_1d<double,3>& An )
{
    Geometry<Node<3> >& pGeometry = this->GetGeometry();

    array_1d<double,3> v1,v2;
    v1[0] = pGeometry[1].X() - pGeometry[0].X();
    v1[1] = pGeometry[1].Y() - pGeometry[0].Y();
    v1[2] = pGeometry[1].Z() - pGeometry[0].Z();

    v2[0] = pGeometry[2].X() - pGeometry[0].X();
    v2[1] = pGeometry[2].Y() - pGeometry[0].Y();
    v2[2] = pGeometry[2].Z() - pGeometry[0].Z();

    MathUtils<double>::CrossProduct(An,v1,v2);
    An *= 0.5;
```

```
}


template class HeatEquationNeumannCondition<2,2>;
template class HeatEquationNeumannCondition<3,3>;

} // namespace Kratos
```

F HeatEquation.cpp

```
//     /  /              /
//     ' /   __/ _` | __/  _ \   __/
//     . \   /   (   | |   (   /\__ `
//    _/\_\_/   \__,_/\__/\___/ ____/
//                      Multi-Physics
//
//  License:              BSD License
//       Kratos default license: kratos/license.txt
//
//  Main authors:     Christian Rossi
//

#include "custom_elements/heat_equation.h"

namespace Kratos {

  template<>
  void HeatEquation<3>::EquationIdVector
  (EquationIdVectorType& rResult, ProcessInfo& rCurrentProcessInfo)
  {
      KRATOS_TRY

      unsigned int Dim = 3;
      unsigned int NumNodes = 4;
      unsigned int DofSize  = NumNodes;

      if (rResult.size() != DofSize)
          rResult.resize(DofSize, false);

      for(unsigned int i=0; i<NumNodes; i++)
      {

          rResult[i]  =  this->GetGeometry()[i].
          GetDof(TEMPERATURE).EquationId();
      }

      KRATOS_CATCH("")
  }
```

```
template<>
void HeatEquation<2>::EquationIdVector
    (EquationIdVectorType& rResult, ProcessInfo& rCurrentProcessInfo)
{
    KRATOS_TRY

    unsigned int Dim = 2;
    unsigned int NumNodes = 3;
    unsigned int DofSize  = NumNodes;

    if (rResult.size() != DofSize)
        rResult.resize(DofSize, false);

    for(unsigned int i=0; i<NumNodes; i++)
    {
        rResult[i] = this->GetGeometry()[i].
        GetDof(TEMPERATURE).EquationId();

    }

    KRATOS_CATCH("")
}


template<>
void HeatEquation<3>::GetDofList
(DofsVectorType& ElementalDofList, ProcessInfo& rCurrentProcessInfo)
{
    KRATOS_TRY

    unsigned int Dim = 3;
    unsigned int NumNodes = 4;
    unsigned int DofSize  = NumNodes;

    if (ElementalDofList.size() != DofSize)
        ElementalDofList.resize(DofSize);

    for(unsigned int i=0; i<NumNodes; i++)
    {
        ElementalDofList[i] = this->GetGeometry()[i].
        pGetDof(TEMPERATURE);

    }

    KRATOS_CATCH("");
}

template<>
void HeatEquation<2>::GetDofList
    (DofsVectorType& ElementalDofList, ProcessInfo& rCurrentProcessInfo)
```

```
{
    KRATOS_TRY

    unsigned int Dim = 2;
    unsigned int NumNodes = 3;
    unsigned int DofSize  = NumNodes;

    if (ElementalDofList.size() != DofSize)
        ElementalDofList.resize(DofSize);

    for(unsigned int i=0; i<NumNodes; i++)
    {

        ElementalDofList[i] = this->GetGeometry()[i].
        pGetDof(TEMPERATURE);
    }

    KRATOS_CATCH("");
}



template<>
void HeatEquation<3>::ComputeGaussPointLHSContribution
     (bounded_matrix<double,4,4>& lhs,
     const ElementDataStruct& data)
{
    const int nnodes = 4;
    const int dim = 3;
    const double rho = data.rho;
    const double k = data.k;
    const double cp = data.cp;

    const double& bdf0 = data.bdf0;
    const double& bdf1 = data.bdf1;
    const double& bdf2 = data.bdf2;
    const double h = data.h;
    const double& delta_t = data.delta_t;
    const array_1d<double,nnodes>& Q = data.Q;
    const bounded_matrix<double,nnodes,dim>& v = data.v;
    const array_1d<double,nnodes>& temp = data.temp;
    const array_1d<double,nnodes>& tempn = data.tempn;
    const array_1d<double,nnodes>& tempnn = data.tempnn;
    const double& dyn_tau_coeff = data.dyn_tau_coeff;

    // Get shape function values
    const array_1d<double,nnodes>& N = data.N;
    const bounded_matrix<double,nnodes,dim>& DN = data.DN_DX;
```

```
    const array_1d<double,dim> v_gauss = prod(trans(v), N);
    const double v_norm = norm_2(v_gauss);
    // Stabilization parameters
    const double c1 = 2.0;
    const double c2 = 4.0;
    const double tau = dyn_tau_coeff*1.0/(rho*cp/delta_t
                     + c1*rho*cp*v_norm/h + c2*k/(h*h));

const double clhs0 =              bdf0*cp*rho;
const double clhs1 =              N[0]*cp*rho;
const double clhs2 =              N[0]*v(0,0) + N[1]*v(1,0) + N[2]*v(2,0)
                                  + N[3]*v(3,0);
const double clhs3 =              N[0]*v(0,1) + N[1]*v(1,1) + N[2]*v(2,1)
                                  + N[3]*v(3,1);
const double clhs4 =              N[0]*v(0,2) + N[1]*v(1,2) + N[2]*v(2,2)
                                  + N[3]*v(3,2);
const double clhs5 =              DN(0,0)*clhs2 + DN(0,1)*clhs3
                                  +DN(0,2)*clhs4;
const double clhs6 =              N[0]*bdf0 + clhs5;
const double clhs7 =              pow(cp, 2);
const double clhs8 =              pow(rho, 2);
const double clhs9 =              clhs5*clhs7*clhs8*tau;
const double clhs10 =             DN(0,0)*k;
const double clhs11 =             DN(0,1)*k;
const double clhs12 =             DN(0,2)*k;
const double clhs13 =             N[0]*bdf0*cp*rho;
const double clhs14 =             DN(1,0)*clhs10 + DN(1,1)*clhs11 +
                                  DN(1,2)*clhs12 + N[1]*clhs13;
const double clhs15 =             DN(1,0)*clhs2 + DN(1,1)*clhs3 +
                                  DN(1,2)*clhs4;
const double clhs16 =             N[1]*bdf0 + clhs15;
const double clhs17 =             DN(2,0)*clhs10 + DN(2,1)*clhs11 +
DN(2,2)*clhs12 + N[2]*clhs13;
const double clhs18 =             DN(2,0)*clhs2 + DN(2,1)*clhs3 +
DN(2,2)*clhs4;
const double clhs19 =             N[2]*bdf0;
const double clhs20 =             clhs18 + clhs19;
const double clhs21 =             DN(3,0)*clhs10 + DN(3,1)*clhs11 +
DN(3,2)*clhs12 + N[3]*clhs13;
const double clhs22 =             DN(3,0)*clhs2 + DN(3,1)*clhs3 +
DN(3,2)*clhs4;
const double clhs23 =             N[3]*bdf0 + clhs22;
const double clhs24 =             N[1]*cp*rho;
const double clhs25 =             clhs15*clhs7*clhs8*tau;
const double clhs26 =             DN(1,0)*k;
const double clhs27 =             DN(1,1)*k;
const double clhs28 =             DN(1,2)*k;
const double clhs29 =             N[1]*bdf0*cp*rho;
const double clhs30 =             DN(2,0)*clhs26 + DN(2,1)*clhs27 +
DN(2,2)*clhs28 + N[2]*clhs29;
```

```
const double clhs31 =              DN(3,0)*clhs26 + DN(3,1)*clhs27 +
DN(3,2)*clhs28 + N[3]*clhs29;
const double clhs32 =              N[2]*cp*rho;
const double clhs33 =              clhs18*clhs7*clhs8*tau;
const double clhs34 =              N[3]*cp*rho;
const double clhs35 =              DN(2,0)*DN(3,0)*k + DN(2,1)*DN(3,1)*k +
DN(2,2)*DN(3,2)*k + clhs19*clhs34;
const double clhs36 =              clhs22*clhs7*clhs8*tau;
            lhs(0,0)=pow(DN(0,0), 2)*k + pow(DN(0,1), 2)*k
            + pow(DN(0,2), 2)*k + pow(N[0], 2)*clhs0 + clhs1*clhs5
            + clhs6*clhs9;
            lhs(0,1)=clhs1*clhs15 + clhs14 + clhs16*clhs9;
            lhs(0,2)=clhs1*clhs18 + clhs17 + clhs20*clhs9;
            lhs(0,3)=clhs1*clhs22 + clhs21 + clhs23*clhs9;
            lhs(1,0)=clhs14 + clhs24*clhs5 + clhs25*clhs6;
            lhs(1,1)=pow(DN(1,0), 2)*k + pow(DN(1,1), 2)*k
            + pow(DN(1,2), 2)*k + pow(N[1], 2)*clhs0
            + clhs15*clhs24 + clhs16*clhs25;
            lhs(1,2)=clhs18*clhs24 + clhs20*clhs25 + clhs30;
            lhs(1,3)=clhs22*clhs24 + clhs23*clhs25 + clhs31;
            lhs(2,0)=clhs17 + clhs32*clhs5 + clhs33*clhs6;
            lhs(2,1)=clhs15*clhs32 + clhs16*clhs33 + clhs30;
            lhs(2,2)=pow(DN(2,0), 2)*k + pow(DN(2,1), 2)*k
            + pow(DN(2,2), 2)*k + pow(N[2], 2)*clhs0
            + clhs18*clhs32 + clhs20*clhs33;
            lhs(2,3)=clhs22*clhs32 + clhs23*clhs33 + clhs35;
            lhs(3,0)=clhs21 + clhs34*clhs5 + clhs36*clhs6;
            lhs(3,1)=clhs15*clhs34 + clhs16*clhs36 + clhs31;
            lhs(3,2)=clhs18*clhs34 + clhs20*clhs36 + clhs35;
            lhs(3,3)=pow(DN(3,0), 2)*k + pow(DN(3,1), 2)*k
            + pow(DN(3,2), 2)*k + pow(N[3], 2)*clhs0
            + clhs22*clhs34 + clhs23*clhs36;


}


template<>
void HeatEquation<2>::ComputeGaussPointLHSContribution
     (bounded_matrix<double,3,3>& lhs, const ElementDataStruct& data)
{
    const int nnodes = 3;
    const int dim = 2;
    const double rho = data.rho;
        const double k = data.k;
    const double cp = data.cp;
    const double h = data.h;
    const double& delta_t = data.delta_t;
    const double& bdf0 = data.bdf0;
    const double& bdf1 = data.bdf1;
```

```
    const double& bdf2 = data.bdf2;
    const array_1d<double,nnodes>& Q = data.Q;
    const bounded_matrix<double,nnodes,dim>& v = data.v;
    const array_1d<double,nnodes>& temp = data.temp;
    const array_1d<double,nnodes>& tempn = data.tempn;
    const array_1d<double,nnodes>& tempnn = data.tempnn;
    const double& dyn_tau_coeff = data.dyn_tau_coeff;



    // Get shape function values
    const array_1d<double,nnodes>& N = data.N;
    const bounded_matrix<double,nnodes,dim>& DN = data.DN_DX;

    // Stabilization parameters
    const array_1d<double,dim> v_gauss = prod(trans(v), N);
    const double v_norm = norm_2(v_gauss);
    // Stabilization parameters
    const double c1 = 2.0;
    const double c2 = 4.0;
    const double tau = dyn_tau_coeff*1.0/(rho*cp/delta_t +
c1*rho*cp*v_norm/h + c2*k/(h*h));

const double clhs0 =              bdf0*cp*rho;
const double clhs1 =              N[0]*cp*rho;
const double clhs2 =              N[0]*v(0,0) + N[1]*v(1,0) + N[2]*v(2,0);
const double clhs3 =              N[0]*v(0,1) + N[1]*v(1,1) + N[2]*v(2,1);
const double clhs4 =              DN(0,0)*clhs2 + DN(0,1)*clhs3;
const double clhs5 =              N[0]*bdf0 + clhs4;
const double clhs6 =              pow(cp, 2);
const double clhs7 =              pow(rho, 2);
const double clhs8 =              clhs4*clhs6*clhs7*tau;
const double clhs9 =              DN(0,0)*k;
const double clhs10 =              DN(0,1)*k;
const double clhs11 =              N[0]*bdf0*cp*rho;
const double clhs12 =              DN(1,0)*clhs9 + DN(1,1)*clhs10
                                   + N[1]*clhs11;
const double clhs13 =              DN(1,0)*clhs2 + DN(1,1)*clhs3;
const double clhs14 =              N[1]*bdf0;
const double clhs15 =              clhs13 + clhs14;
const double clhs16 =              DN(2,0)*clhs9 + DN(2,1)*clhs10
                                   + N[2]*clhs11;
const double clhs17 =              DN(2,0)*clhs2 + DN(2,1)*clhs3;
const double clhs18 =              N[2]*bdf0 + clhs17;
const double clhs19 =              N[1]*cp*rho;
const double clhs20 =              clhs13*clhs6*clhs7*tau;
const double clhs21 =              N[2]*cp*rho;
const double clhs22 =              DN(1,0)*DN(2,0)*k + DN(1,1)*DN(2,1)*k
                                   + clhs14*clhs21;
const double clhs23 =              clhs17*clhs6*clhs7*tau;
```

```
            lhs(0,0)=pow(DN(0,0), 2)*k + pow(DN(0,1), 2)*k
            + pow(N[0], 2)*clhs0 + clhs1*clhs4 + clhs5*clhs8;
            lhs(0,1)=clhs1*clhs13 + clhs12 + clhs15*clhs8;
            lhs(0,2)=clhs1*clhs17 + clhs16 + clhs18*clhs8;
            lhs(1,0)=clhs12 + clhs19*clhs4 + clhs20*clhs5;
            lhs(1,1)=pow(DN(1,0), 2)*k + pow(DN(1,1), 2)*k
            + pow(N[1], 2)*clhs0 + clhs13*clhs19 + clhs15*clhs20;
            lhs(1,2)=clhs17*clhs19 + clhs18*clhs20 + clhs22;
            lhs(2,0)=clhs16 + clhs21*clhs4 + clhs23*clhs5;
            lhs(2,1)=clhs13*clhs21 + clhs15*clhs23 + clhs22;
            lhs(2,2)=pow(DN(2,0), 2)*k + pow(DN(2,1), 2)*k
            + pow(N[2], 2)*clhs0 + clhs17*clhs21 + clhs18*clhs23;


}


template<>
void HeatEquation<3>::ComputeGaussPointRHSContribution
     (array_1d<double,4>& rhs, const ElementDataStruct& data)
{
    const int nnodes = 4;
    const int dim = 3;
    const double rho = data.rho;
    const double k = data.k;
    const double cp = data.cp;
    const double h = data.h;
    const double& delta_t = data.delta_t;
    const double& bdf0 = data.bdf0;
    const double& bdf1 = data.bdf1;
    const double& bdf2 = data.bdf2;
    const array_1d<double,nnodes>& Q = data.Q;
    const bounded_matrix<double,nnodes,dim>& v = data.v;
    const array_1d<double,nnodes>& temp = data.temp;
    const array_1d<double,nnodes>& tempn = data.tempn;
    const array_1d<double,nnodes>& tempnn = data.tempnn;
    const double& dyn_tau_coeff = data.dyn_tau_coeff;

    // Get shape function values
    const array_1d<double,nnodes>& N = data.N;
    const bounded_matrix<double,nnodes,dim>& DN = data.DN_DX;

    // Auxiliary variables used in the calculation of the RHS
    const double q_gauss = inner_prod(data.Q, data.N);

    // Stabilization parameters
    const array_1d<double,dim> v_gauss = prod(trans(v), N);
    const double v_norm = norm_2(v_gauss);
    // Stabilization parameters
    const double c1 = 2.0;
```

```
    const double c2 = 4.0;
    const double tau = dyn_tau_coeff*1.0/(rho*cp/delta_t +
                       c1*rho*cp*v_norm/h + c2*k/(h*h));


const double crhs0 =              N[0]*Q[0] + N[1]*Q[1] + N[2]*Q[2]
                                  + N[3]*Q[3];
const double crhs1 =              DN(0,0)*temp[0] + DN(1,0)*temp[1]
                                  + DN(2,0)*temp[2] + DN(3,0)*temp[3];
const double crhs2 =              crhs1*k;
const double crhs3 =              DN(0,1)*temp[0] + DN(1,1)*temp[1]
                                  + DN(2,1)*temp[2] + DN(3,1)*temp[3];
const double crhs4 =              crhs3*k;
const double crhs5 =              DN(0,2)*temp[0] + DN(1,2)*temp[1]
                                  + DN(2,2)*temp[2] + DN(3,2)*temp[3];
const double crhs6 =              crhs5*k;
const double crhs7 =              N[0]*(bdf0*temp[0] + bdf1*tempn[0]
                                  + bdf2*tempnn[0]) + N[1]*(bdf0*temp[1]
                                  + bdf1*tempn[1] + bdf2*tempnn[1]) +
                                  N[2]*(bdf0*temp[2] + bdf1*tempn[2] +
                                  bdf2*tempnn[2]) + N[3]*(bdf0*temp[3] +
                                  bdf1*tempn[3] + bdf2*tempnn[3]);
const double crhs8 =              cp*crhs7*rho;
const double crhs9 =              N[0]*v(0,0) + N[1]*v(1,0) + N[2]*v(2,0)
                                  + N[3]*v(3,0);
const double crhs10 =             N[0]*v(0,1) + N[1]*v(1,1) + N[2]*v(2,1)
                                  + N[3]*v(3,1);
const double crhs11 =             N[0]*v(0,2) + N[1]*v(1,2) + N[2]*v(2,2)
                                  + N[3]*v(3,2);
const double crhs12 =             crhs1*crhs9 + crhs10*crhs3 +
crhs11*crhs5;
const double crhs13 =             cp*crhs12*rho;
const double crhs14 =             cp*rho*tau*(-cp*rho*(crhs12 + crhs7)
                                   + crhs0);
            rhs[0]=-DN(0,0)*crhs2 - DN(0,1)*crhs4 - DN(0,2)*crhs6
            + N[0]*crhs0 - N[0]*crhs13 - N[0]*crhs8 + crhs14*(DN(0,0)
            *crhs9 + DN(0,1)*crhs10 + DN(0,2)*crhs11);
            rhs[1]=-DN(1,0)*crhs2 - DN(1,1)*crhs4 - DN(1,2)*crhs6
            + N[1]*crhs0 - N[1]*crhs13 - N[1]*crhs8 + crhs14*(DN(1,0)
            *crhs9 + DN(1,1)*crhs10 + DN(1,2)*crhs11);
            rhs[2]=-DN(2,0)*crhs2 - DN(2,1)*crhs4 - DN(2,2)*crhs6
            + N[2]*crhs0 - N[2]*crhs13 - N[2]*crhs8 + crhs14*(DN(2,0)
            *crhs9 + DN(2,1)*crhs10 + DN(2,2)*crhs11);
            rhs[3]=-DN(3,0)*crhs2 - DN(3,1)*crhs4 - DN(3,2)*crhs6
            + N[3]*crhs0 - N[3]*crhs13 - N[3]*crhs8 + crhs14*(DN(3,0)
            *crhs9 + DN(3,1)*crhs10 + DN(3,2)*crhs11);

}
```

```
template<>
void HeatEquation<2>::ComputeGaussPointRHSContribution
     (array_1d<double,3>& rhs, const ElementDataStruct& data)
{
    const int nnodes = 3;
    const int dim = 2;
    const double rho = data.rho;
    const double k = data.k;
    const double cp = data.cp;
    const double h = data.h;
    const double& delta_t = data.delta_t;
    const double& bdf0 = data.bdf0;
    const double& bdf1 = data.bdf1;
    const double& bdf2 = data.bdf2;
    const array_1d<double,nnodes>& Q = data.Q;
    const bounded_matrix<double,nnodes,dim>& v = data.v;
    const array_1d<double,nnodes>& temp = data.temp;
    const array_1d<double,nnodes>& tempn = data.tempn;
    const array_1d<double,nnodes>& tempnn = data.tempnn;
    const double& dyn_tau_coeff = data.dyn_tau_coeff;

    // Get shape function values
    const array_1d<double,nnodes>& N = data.N;
    const bounded_matrix<double,nnodes,dim>& DN = data.DN_DX;


    const double q_gauss = inner_prod(data.Q, data.N);

    // Stabilization parameters
    const array_1d<double,dim> v_gauss = prod(trans(v), N);
    const double v_norm = norm_2(v_gauss);
     Stabilization parameters
    const double c1 = 2.0;
    const double c2 = 4.0;
    const double tau = dyn_tau_coeff*1.0/(rho*cp/delta_t
                     + c1*rho*cp*v_norm/h + c2*k/(h*h));
    const double crhs0 =             N[0]*Q[0] + N[1]*Q[1] + N[2]*Q[2];
const double crhs1 =           DN(0,0)*temp[0] + DN(1,0)*temp[1] +
DN(2,0)*temp[2];
const double crhs2 =           crhs1*k;
const double crhs3 =           DN(0,1)*temp[0] + DN(1,1)*temp[1] +
DN(2,1)*temp[2];
const double crhs4 =           crhs3*k;
const double crhs5 =           N[0]*(bdf0*temp[0] + bdf1*tempn[0]
                               + bdf2*tempnn[0]) + N[1]*(bdf0*temp[1]
                               + bdf1*tempn[1] + bdf2*tempnn[1])
                               + N[2]*(bdf0*temp[2] + bdf1*tempn[2]
                               + bdf2*tempnn[2]);
const double crhs6 =           cp*crhs5*rho;
const double crhs7 =           N[0]*v(0,0) + N[1]*v(1,0) + N[2]*v(2,0);
```

```
const double crhs8 =                N[0]*v(0,1) + N[1]*v(1,1) + N[2]*v(2,1);
const double crhs9 =                crhs1*crhs7 + crhs3*crhs8;
const double crhs10 =               cp*crhs9*rho;
const double crhs11 =               cp*rho*tau*(-cp*rho*(crhs5 + crhs9)
                                    + crhs0);
            rhs[0]=-DN(0,0)*crhs2 - DN(0,1)*crhs4 + N[0]*crhs0
            - N[0]*crhs10 - N[0]*crhs6 + crhs11*(DN(0,0)*crhs7
            + DN(0,1)*crhs8);
            rhs[1]=-DN(1,0)*crhs2 - DN(1,1)*crhs4 + N[1]*crhs0
            - N[1]*crhs10 - N[1]*crhs6 + crhs11*(DN(1,0)*crhs7
            + DN(1,1)*crhs8);
            rhs[2]=-DN(2,0)*crhs2 - DN(2,1)*crhs4 + N[2]*crhs0
            - N[2]*crhs10 - N[2]*crhs6 + crhs11*(DN(2,0)*crhs7
            + DN(2,1)*crhs8);


}
```

G HeatEquation.h

```
    |  /           |
    '/ _ \ _ | _ \ _ \ _ \ _ |
 _ \ / _ \ _ | _ \ ( _ | _ \ ( _ | \ _ \
 _ | \ _ \ _ | _ \ _ , _ | \ _ | \ _ / \ _ _ _ /
 _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ Multi-Physics
//
//  License:  BSD License
//  Kratos default license: kratos/license.txt
//
//  Main authors:  Christian Rossi
//

#if !defined(KRATOS_HEAT_EQUATION)
#define  KRATOS_HEAT_EQUATION

// System includes

// External includes
#include "boost/smart_ptr.hpp"

// Project includes
#include "includes/define.h"
#include "includes/element.h"
#include "includes/ublas_interface.h"
#include "includes/variables.h"
#include "includes/serializer.h"
#include "utilities/geometry_utilities.h"

// Application includes
```

```
namespace Kratos
{

///@name Kratos Globals
///@{

///@}
///@name Type Definitions
///@{

///@}
///@name  Enum's
///@{

///@}
///@name  Functions
///@{

///@}
///@name Kratos Classes
///@{


*/
template< unsigned int TDim, unsigned int TNumNodes = TDim + 1 >
class HeatEquation : public Element
{
public:
    ///@name Type Definitions
    ///@{

    /// Counted pointer of
    KRATOS_CLASS_POINTER_DEFINITION(HeatEquation);

    struct ElementDataStruct
    {
        bounded_matrix<double, TNumNodes, TDim> v;
        array_1d<double,TNumNodes> temp, tempn, tempnn, Q;

        bounded_matrix<double, TNumNodes, TDim > DN_DX;
        array_1d<double, TNumNodes > N;
              double rho;
        double cp;
        double k;
        double bdf0;
        double bdf1;
        double bdf2;
        double h;
        double volume;
        double delta_t;
```

```
    double dyn_tau_coeff;
};

///@}
///@name Life Cycle
///@{

/// Default constructor.

HeatEquation(IndexType NewId, GeometryType::Pointer pGeometry)
: Element(NewId, pGeometry)
{}

HeatEquation(IndexType NewId, GeometryType::Pointer pGeometry,
             PropertiesType::Pointer pProperties)
             : Element(NewId, pGeometry, pProperties)
{}

/// Destructor.
virtual ~HeatEquation() {};


///@}
///@name Operators
///@{


///@}
///@name Operations
///@{

Element::Pointer Create
(IndexType NewId, NodesArrayType const& rThisNodes,
PropertiesType::Pointer pProperties) const override
{
    KRATOS_TRY
    return boost::make_shared< HeatEquation < TDim, TNumNodes >

    >(NewId, this->GetGeometry().Create(rThisNodes), pProperties);
    KRATOS_CATCH("");
}

Element::Pointer Create(IndexType NewId, GeometryType
       ::Pointer pGeom, PropertiesType::Pointer pProperties)
       const override
{
    KRATOS_TRY
    return boost::make_shared< HeatEquation < TDim, TNumNodes >
    >(NewId, pGeom, pProperties);
    KRATOS_CATCH("");
```

```
    }


    void CalculateLocalSystem(MatrixType& rLeftHandSideMatrix,
                              VectorType& rRightHandSideVector,
                              ProcessInfo& rCurrentProcessInfo)
                              override
    {
        KRATOS_TRY



        constexpr unsigned int MatrixSize = TNumNodes;

        if (rLeftHandSideMatrix.size1() != MatrixSize)
            rLeftHandSideMatrix.resize(MatrixSize, MatrixSize, false);

        if (rRightHandSideVector.size() != MatrixSize)
            rRightHandSideVector.resize(MatrixSize, false);


        // Struct to pass around the data
        ElementDataStruct data;
        this->FillElementData(data, rCurrentProcessInfo);

        // Allocate memory needed
        bounded_matrix<double,MatrixSize, MatrixSize> lhs_local;
        array_1d<double,MatrixSize> rhs_local;

        // Loop on gauss points
        noalias(rLeftHandSideMatrix) = ZeroMatrix(MatrixSize,MatrixSize);
        noalias(rRightHandSideVector) = ZeroVector(MatrixSize);



        // Gauss point position
        bounded_matrix<double,TNumNodes, TNumNodes> Ncontainer;
        GetShapeFunctionsOnGauss(Ncontainer);


        for(unsigned int igauss = 0; igauss<Ncontainer.size1();
            igauss++)
        {
            noalias(data.N) = row(Ncontainer, igauss);


            ComputeGaussPointLHSContribution(lhs_local, data);
            ComputeGaussPointRHSContribution(rhs_local, data);

            noalias(rLeftHandSideMatrix) += lhs_local;
```

```
            noalias(rRightHandSideVector) += rhs_local;

        }


        rLeftHandSideMatrix  *= data.volume/
                                static_cast<double>(TNumNodes);
        rRightHandSideVector *= data.volume/
                                static_cast<double>(TNumNodes);

        KRATOS_CATCH("")
    }



    void CalculateRightHandSide(VectorType& rRightHandSideVector,
                        ProcessInfo& rCurrentProcessInfo) override
    {
        KRATOS_TRY


        constexpr unsigned int MatrixSize = TNumNodes;

        if (rRightHandSideVector.size() != MatrixSize)
            rRightHandSideVector.resize(MatrixSize, false);

        // Struct to pass around the data
        ElementDataStruct data;
        this->FillElementData(data, rCurrentProcessInfo);

        // Allocate memory needed
        array_1d<double,MatrixSize> rhs_local;

        // Gauss point position
        bounded_matrix<double,TNumNodes, TNumNodes> Ncontainer;
        GetShapeFunctionsOnGauss(Ncontainer);

        // Loop on gauss point
        noalias(rRightHandSideVector) = ZeroVector(MatrixSize);
        for(unsigned int igauss = 0; igauss<Ncontainer.size2();
            igauss++)
        {
            noalias(data.N) = row(Ncontainer, igauss);

            ComputeGaussPointRHSContribution(rhs_local, data);

            noalias(rRightHandSideVector) += rhs_local;
        }
```

```
        rRightHandSideVector *= data.volume/
                                static_cast<double>(TNumNodes);

        KRATOS_CATCH("")

    }



    virtual int Check(const ProcessInfo& rCurrentProcessInfo) override
    {
        KRATOS_TRY

        // Perform basic element checks
        int ErrorCode = Kratos::Element::Check(rCurrentProcessInfo);
        if(ErrorCode != 0) return ErrorCode;



        if(TAU.Key() == 0)
           KRATOS_THROW_ERROR(std::invalid_argument,
           "TAU Key is 0.
            Check if the application was correctly registered.","");
        if(DENSITY.Key() == 0)
           KRATOS_THROW_ERROR(std::invalid_argument,
           "DENSITY Key is 0.
            Check if the application was correctly registered.","");
        if(CONDUCTIVITY.Key() == 0)
            KRATOS_THROW_ERROR(std::invalid_argument,
            "CONDUCTIVITY Key is 0.
            Check if the application was correctly registered.","");
        if(SPECIFIC_HEAT.Key() == 0)
            KRATOS_THROW_ERROR(std::invalid_argument,
            "SPECIFIC_HEAT Key is 0.
            Check if the application was correctly registered.","");

        for(unsigned int i=0; i<this->GetGeometry().size(); ++i)
        {
            //if(this->GetGeometry()[i].SolutionStepsDataHas(VELOCITY)
            == false)
            //     KRATOS_THROW_ERROR(std::invalid_argument,
            "Missing VELOCITY variable on solution step data for node
            ",this->GetGeometry()[i].Id());
            if(this->GetGeometry()[i].SolutionStepsDataHas(TEMPERATURE)
            == false)
            KRATOS_THROW_ERROR(std::invalid_argument,
            "Missing TEMPERATURE variable on solution step data for node
            ",this->GetGeometry()[i].Id());
```

```
            if(this->GetGeometry()[i].HasDofFor(TEMPERATURE) == false)
                KRATOS_THROW_ERROR(std::invalid_argument,
                "Missing TEMPERATURE component degree of freedom on node
                ",this->GetGeometry()[i].Id());
        }

            KRATOS_CATCH("");
    }


      virtual std::string Info() const override
    {
        return "HeatEquation #";
    }

    /// Print information about this object.
    virtual void PrintInfo(std::ostream& rOStream) const override
    {
        rOStream << Info() << Id();
    }

    /// Print object's data.
    // virtual void PrintData(std::ostream& rOStream) const override

    ///@}
    ///@name Friends
    ///@{


    ///@}

protected:
    ///@name Protected static member variables
    ///@{


    ///@}
    ///@name Protected member Variables
    ///@{

    // Constitutive law pointer
    // ConstitutiveLaw::Pointer mpConstitutiveLaw = nullptr;

    // Symbolic function implementing the element
        GetDofList<TDim,TNumNodes>
    void GetDofList(DofsVectorType& ElementalDofList,
         ProcessInfo& rCurrentProcessInfo) override;
    void EquationIdVector(EquationIdVectorType& rResult,
         ProcessInfo& rCurrentProcessInfo) override;
```

```
    void ComputeGaussPointLHSContribution
    (bounded_matrix<double,TNumNodes,TNumNodes>& lhs,
    const ElementDataStruct& data);
    void ComputeGaussPointRHSContribution
    (array_1d<double,TNumNodes>& rhs,
    const ElementDataStruct& data);

    //double SubscaleErrorEstimate(const ElementDataStruct& data);

    ///@}
    ///@name Protected Operators
    ///@{

    HeatEquation() : Element()
    {
    }


    // Auxiliar function to fill the element data structure
    void FillElementData(ElementDataStruct& rData,
    const ProcessInfo& rCurrentProcessInfo)
    {



        GeometryUtils::CalculateGeometryData(this->GetGeometry(),
        rData.DN_DX, rData.N, rData.volume);

        // Compute element size
        rData.h = ComputeH(rData.DN_DX);

        // Database access to all of the variables needed
        const Vector& BDFVector = rCurrentProcessInfo[BDF_COEFFICIENTS];

        rData.delta_t = rCurrentProcessInfo[DELTA_TIME];

        rData.bdf0 = BDFVector[0];
        rData.bdf1 = BDFVector[1];
        rData.bdf2 = BDFVector[2];


        rData.dyn_tau_coeff = rCurrentProcessInfo[TAU];
        rData.cp = this->GetProperties()[SPECIFIC_HEAT];
        rData.rho = this->GetProperties()[DENSITY];
        rData.k = this->GetProperties()[CONDUCTIVITY];


        for (unsigned int i = 0; i < TNumNodes; i++)
        {
```

```
            const array_1d<double,3>& vel = this->GetGeometry()[i].
            FastGetSolutionStepValue(VELOCITY);

            for(unsigned int k=0; k<TDim; k++)
            {
                rData.v(i,k)    = vel[k];

            }

            rData.Q[i] = this->GetGeometry()[i].
            FastGetSolutionStepValue(HEAT_FLUX);
            rData.temp[i] = this->GetGeometry()[i].
            FastGetSolutionStepValue(TEMPERATURE);
            rData.tempn[i] = this->GetGeometry()[i].
            FastGetSolutionStepValue(TEMPERATURE,1);
            rData.tempnn[i] = this->GetGeometry()[i].
            FastGetSolutionStepValue(TEMPERATURE,2);
        }

    }

    //~ template< unsigned int TDim, unsigned int TNumNodes=TDim+1>
    double ComputeH(boost::numeric::ublas::bounded_matrix<double
    ,TNumNodes, TDim>& DN_DX)
    {
        double h=0.0;
        for(unsigned int i=0; i<TNumNodes; i++)
        {
            double h_inv = 0.0;
            for(unsigned int k=0; k<TDim; k++)
            {
                h_inv += DN_DX(i,k)*DN_DX(i,k);
            }
            h += 1.0/h_inv;
        }
        h = sqrt(h)/static_cast<double>(TNumNodes);
        return h;
    }

    // 3D tetrahedra shape functions values at Gauss points
    void GetShapeFunctionsOnGauss(boost::numeric::ublas::
         bounded_matrix<double,4,4>& Ncontainer)
    {
        Ncontainer(0,0) = 0.58541020;
        Ncontainer(0,1) = 0.13819660;
        Ncontainer(0,2) = 0.13819660;
        Ncontainer(0,3) = 0.13819660;
        Ncontainer(1,0) = 0.13819660;
        Ncontainer(1,1) = 0.58541020;
        Ncontainer(1,2) = 0.13819660;
```

```
        Ncontainer(1,3) = 0.13819660;
        Ncontainer(2,0) = 0.13819660;
        Ncontainer(2,1) = 0.13819660;
        Ncontainer(2,2) = 0.58541020;
        Ncontainer(2,3) = 0.13819660;
        Ncontainer(3,0) = 0.13819660;
        Ncontainer(3,1) = 0.13819660;
        Ncontainer(3,2) = 0.13819660;
        Ncontainer(3,3) = 0.58541020;
    }

    // 2D triangle shape functions values at Gauss points
    void GetShapeFunctionsOnGauss(boost::numeric::ublas::
         bounded_matrix<double,3,3>& Ncontainer)
    {
        const double one_sixt = 1.0/6.0;
        const double two_third = 2.0/3.0;
        Ncontainer(0,0) = one_sixt;
        Ncontainer(0,1) = one_sixt;
        Ncontainer(0,2) = two_third;
        Ncontainer(1,0) = one_sixt;
        Ncontainer(1,1) = two_third;
        Ncontainer(1,2) = one_sixt;
        Ncontainer(2,0) = two_third;
        Ncontainer(2,1) = one_sixt;
        Ncontainer(2,2) = one_sixt;
    }

    // 3D tetrahedra shape functions values at centered Gauss point
    void GetShapeFunctionsOnUniqueGauss(boost::numeric::
         ublas::bounded_matrix<double,1,4>& Ncontainer)
    {
        Ncontainer(0,0) = 0.25;
        Ncontainer(0,1) = 0.25;
        Ncontainer(0,2) = 0.25;
        Ncontainer(0,3) = 0.25;
    }

    // 2D triangle shape functions values at centered Gauss point
    void GetShapeFunctionsOnUniqueGauss(boost::numeric::
         ublas::bounded_matrix<double,1,3>& Ncontainer)
    {
        Ncontainer(0,0) = 1.0/3.0;
        Ncontainer(0,1) = 1.0/3.0;
        Ncontainer(0,2) = 1.0/3.0;
    }

    ///@}
    ///@name Protected  Access
    ///@{
```

```
    ///@}
    ///@name Protected Inquiry
    ///@{


    ///@}
    ///@name Protected LifeCycle
    ///@{


    ///@}
private:
    ///@name Static Member Variables
    ///@{

    ///@}
    ///@name Member Variables
    ///@{

    ///@}
    ///@name Serialization
    ///@{
    friend class Serializer;

    virtual void save(Serializer& rSerializer) const override
    {
        KRATOS_SERIALIZE_SAVE_BASE_CLASS(rSerializer, Element);
    }

    virtual void load(Serializer& rSerializer) override
    {
        KRATOS_SERIALIZE_LOAD_BASE_CLASS(rSerializer, Element);
    }

    ///@}

    ///@name Private Operations
    ///@{


    ///@}
    ///@name Private  Access
    ///@{


    ///@}
    ///@name Private Inquiry
    ///@{
```

```
    ///@}
    ///@name Un accessible methods
    ///@{

    ///@}

};

///@}

///@name Type Definitions
///@{


///@}
///@name Input and output
///@{


/// input stream function
/*  inline std::istream& operator >> (std::istream& rIStream,
                                      Fluid2DASGS& rThis);
 */
/// output stream function
/*  inline std::ostream& operator << (std::ostream& rOStream,
                                      const Fluid2DASGS& rThis)
    {
      rThis.PrintInfo(rOStream);
      rOStream << std::endl;
      rThis.PrintData(rOStream);

      return rOStream;
    }*/
///@}

} // namespace Kratos.

#endif //
```

Bibliography

[1] Manzán, M; Oñate, E. *Stabilization Techniques for Finite Element Analysis of Convection-Diffusion Problems of convection-diffusion problems*, 2000.

[2] Moran, M; Shapiro H N.*Engineering Thermodynamics*, 2006.

[3] Schafer, M; Turek, S.*Benchmark Computations of Laminar Flow Around a Cylinder*, 1996.

[4] Cotela, J.*Applications of Turbulence Modeling in Civil Engineering*, 2016.

[5] Kundu, P K. *Fluid Mechanics*, 2012.

[6] Callister, W ;Rethwisch, D. Material Science and Engineering, 2012.

[7] Donea, J; Huerta, A. *Finite Element Methods for Flow Problems*, 2003.

[8] John, V; Tambulea, A. *on Finite Element Variational Multiscale Methods for Incompressible Turbulent Flows*, 2006.

[9] Layton, W; Nathaniel N; Monika, T. *Numerical Analysis of Modular Regularization Methods for the Navier Stokes Equations*, 2013.

[10] John, V; Kindl, A. *A Variational Multiscale Method for Turbulent Flow Simulation with Adaptive Large Scale Space*, 2010.

[11] Cengel, YA. *Heat and Mass Transfer. A Practical Approach*, 2006.

[12] Fay, J. *Introduction to Fluid Mechanics Copyrighted Material Introduction to Fluid Mechanics*, 2004.

[13] Ambrosio, J. *Multibody Dynamics: Bridging for Multidisciplinary Applications*, 2004.

[14] Reddy, J. *The Finite Element Method in Heat Transfer and Fluid Dynamics*, 2017.

[15] Riera Santacreu, M. *Estudi Experimental i Numeric d'un Bescanviador de Calor*, 2017.

[16] Skare, B.*Numerical Methods for Solving Complex Heat Exchanger Models in Transient Operation*, 2014.

[17] Oñate, E. *Introduction to the Finite Element Method*, 2008.

[18] Murthy, J. *Numerical Methods in Heat , Mass and Momentum Transfer*, 2002.

[19] Suhas, V. *Numerical Heat Transfer and Fluid Flow*, 1990.

[20] Hughes, T ;Calo, V ; Scovazzi, G. *Variational and Multiscale Methods in Turbulence*, 2005.

[21] Israel, U. *Implementation of an Algorithm for Data Transfer on the Fluid-Structure Interface between Non-Matching Meshes in Kratos and an Algorithm for the Generation of an Interface between GiD and Kratos*, 2006.

[22] Kuzmin, D. *A Guide to Numerical Methods for Transport Equations*, 2010.

[23] Holman, J; Lloyd, J ; *Fluid Mechanics*, 2010.

[24] Ursula, R, *Computational Multiscale Methods for Turbulent Single and Two-Phase Flows*, 2013.

[25] Hughes, T ;Franca, L; Scovazzi, G, *Multiscale and Stabilized Methods*, 2004.